

OAuth  
Internet-Draft  
Intended status: Standards Track  
Expires: December 2, 2017

W. Denniss  
Google  
J. Bradley  
Ping Identity  
M. Jones  
Microsoft  
H. Tschofenig  
ARM Limited  
May 31, 2017

OAuth 2.0 Device Flow for Browserless and Input Constrained Devices  
draft-ietf-oauth-device-flow-06

Abstract

This OAuth 2.0 authorization flow for browserless and input constrained devices, often referred to as the device flow, enables OAuth clients to request user authorization from devices that have an Internet connection, but don't have an easy input method (such as a smart TV, media console, picture frame, or printer), or lack a suitable browser for a more traditional OAuth flow. This authorization flow instructs the user to perform the authorization request on a secondary device, such as a smartphone. There is no requirement for communication between the constrained device and the user's secondary device.

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 2, 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 . . . . .	3
2. Terminology . . . . .	5
3. Protocol . . . . .	5
3.1. Device Authorization Request . . . . .	5
3.2. Device Authorization Response . . . . .	6
3.3. User Interaction . . . . .	7
3.3.1. Optimization for Non-textual Verification URIs . . . . .	8
3.4. Device Access Token Request . . . . .	8
3.5. Device Access Token Response . . . . .	9
4. Discovery Metadata . . . . .	10
5. Security Considerations . . . . .	10
5.1. User Code Brute Forcing . . . . .	11
5.2. Device Trustworthiness . . . . .	11
5.3. Remote Phishing . . . . .	11
5.4. Non-confidential Clients . . . . .	11
5.5. Non-Visual Code Transmission . . . . .	12
6. Usability Considerations . . . . .	12
6.1. User Code Recommendations . . . . .	12
6.2. Non-Browser User Interaction . . . . .	12
7. IANA Considerations . . . . .	13
7.1. OAuth URI Registration . . . . .	13
7.1.1. Registry Contents . . . . .	13
7.2. OAuth Extensions Error Registration . . . . .	13
7.2.1. Registry Contents . . . . .	13
7.3. OAuth 2.0 Authorization Server Metadata . . . . .	14
7.3.1. Registry Contents . . . . .	14
8. Normative References . . . . .	14
Appendix A. Acknowledgements . . . . .	15
Appendix B. Document History . . . . .	15
Authors' Addresses . . . . .	16

## 1. Introduction

This OAuth 2.0 protocol flow for browserless and input constrained devices, often referred to as the device flow, enables OAuth clients to request user authorization from devices that have an internet connection, but don't have an easy input method (such as a smart TV, media console, picture frame, or printer), or lack a suitable browser for a more traditional OAuth flow. This authorization flow instructs the user to perform the authorization request on a secondary device, such as a smartphone.

The device flow 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 OAuth 2.0 for Native Apps [I-D.ietf-oauth-native-apps].

The only requirements to use this flow are that the device is connected to the Internet, and able to make outbound HTTPS requests, be able to display or otherwise communicate a URI and code sequence to the user, and that the user has a secondary device (e.g., personal computer or smartphone) from which to process the request. There is no requirement for two-way communication between the OAuth client and the user-agent, enabling a broad range of use-cases.

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 client cannot receive incoming requests, it polls the authorization server repeatedly until the end-user completes the approval process.

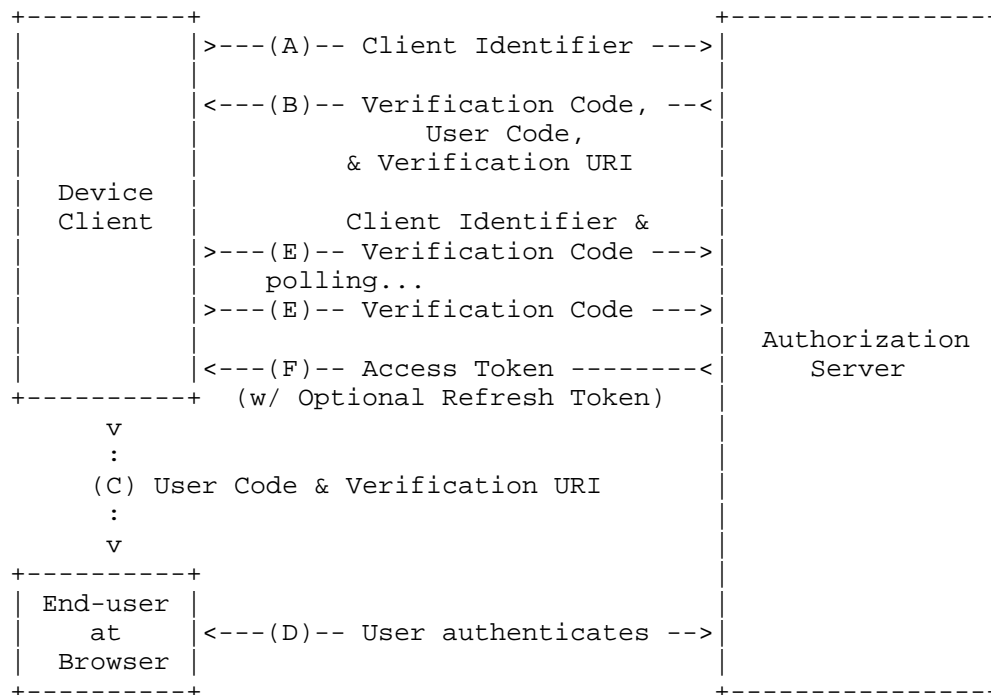


Figure 1: Device Flow.

The device 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 verification code, an end-user code, and provides the end-user verification URI.

(C) The client instructs the end-user to use its user-agent (elsewhere) and visit the provided end-user verification URI. The client provides the end-user with the end-user code to enter in order to grant access.

(D) The authorization server authenticates the end-user (via the user-agent) and prompts the end-user to grant the client's access request. If the end-user agrees to the client's access request, the end-user enters the end-user code provided by the client. The authorization server validates the end-user code provided by the end-user.

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

(F) Assuming the end-user granted access, the authorization server validates the verification code provided by the client and responds back with the access token.

## 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 [RFC2119].

### 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 end-user on the authorization server, and is thus used to bind the device to the end-user.

## 3. Protocol

### 3.1. Device Authorization Request

The client initiates the 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, encoded with the "application/x-www-form-urlencoded" content type:

#### client\_id

REQUIRED. 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 (line breaks are for display purposes only):

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

client_id=459691054427
```

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.

### 3.2. Device Authorization Response

In response, the authorization server generates a 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 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.

**expires\_in**  
OPTIONAL. 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.

For example:

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

{
  "device_code": "GMMhmHCXhWEzkobqIHGG_EnNYYSAkukHspeYUk9E8",
  "user_code": "WDJB-MJHT",
  "verification_uri": "https://www.example.com/device",
  "expires_in": 1800,
  "interval": 5
}
```

### 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 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 informs 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.

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, and is out of scope of this specification.

### 3.3.1. Optimization for Non-textual Verification URIs

Clients MAY present the verification URI in a non-textual manner using any method that results in the browser being opened with the URI, such as with QR codes, or NFC, to save the user typing the URI. For usability reasons, it is RECOMMENDED for clients to still display the unmodified verification URI for users not able to use such a shortcut.

To optimize the user interaction for such non-textual verification URI transmission, clients MAY include the user code as part of the verification URI using the URI parameter "user\_code".

An example verification URI with the user code included:

```
https://example.com/device?user_code=WDJB-MJHT
```

When the user code is included in the verification URI in this way, it is considered as a hint to the authorization server to enable potential optimizations. The authorization server MAY use this hint to optimize the user interaction (such as by removing the need for the user to type the code), it MAY also ignore it completely. The client MUST still display the user code textually, for authorization servers that require users to input the user code manually, or otherwise use the user code as part of a visual confirmation step.

This optimization is intended for non-textual transmission of the verification URI, it is NOT RECOMMENDED to include the user code in verification URIs shown textually, as this increases the length and complexity of the URI that the user must type.

### 3.4. Device Access Token Request

After displaying instructions to the user, the client makes an Access Token Request to the token endpoint 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]) 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].

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=GMMhmHCXhWEzkobqIHGG_EnNYYSakukHspeYUk9E8
&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 5.4.

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 specific for the device flow:

**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.

**slow\_down**

The client is polling too quickly and should back off at a reasonable rate.

#### expired\_token

The "device\_code" has expired. The client will need to make a new Device Authorization Request.

The error codes "authorization\_pending" and "slow\_down" are considered soft errors. The client should continue to poll the token endpoint by repeating the Device Token Request (Section 3.4) when receiving soft errors, increasing the time between polls if a "slow\_down" error is received. Other error codes are considered hard errors; the client should stop polling and react accordingly, for example, by displaying an error to the user.

If the verification codes have expired, the server SHOULD respond with the standard OAuth error "invalid\_grant". Clients MAY then choose to start a new device authorization session.

The interval at which the client polls MUST NOT be more frequent than the "interval" parameter returned in the Device Authorization Response (see Section 3.2).

The assumption of this specification is that the secondary device the user is authorizing the request on does not have a way to communicate back to the OAuth client. Only a one-way channel is required to make this flow useful in many scenarios. For example, an HTML application on a TV that can only make 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. Such behavior is, however, outside the scope of this specification.

## 4. Discovery Metadata

Support for the device flow MAY be declared in the OAuth 2.0 Authorization Server Metadata [I-D.ietf-oauth-discovery] with the following metadata:

#### 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, the entropy is typically less than would be used for the device code or other OAuth bearer token types. 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 makes a brute-force attack infeasible.

### 5.2. 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.

### 5.3. Remote Phishing

It is possible for the device flow to be initiated on a device in an attacker's possession. For example, the 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.

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.

### 5.4. Non-confidential Clients

Most device clients are incapable of being confidential clients, as secrets that are statically included as part of an app distributed to multiple users cannot be considered confidential. For such clients,

the recommendations of Section 5.3.1 of [RFC6819] and Section 8.9 of [I-D.ietf-oauth-native-apps] apply.

#### 5.5. 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, or within range of a short-range wireless signal.

### 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 creation valid 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, with an entropy of  $20^8$ , is "WDJB-MJHT".

The server should ignore any characters like punctuation that are not in the user-code character set. Provided that the character set doesn't include characters of different case, the comparison should be case insensitive.

#### 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 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: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 ]]

### 7.2. OAuth Extensions Error Registration

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

#### 7.2.1. 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: 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 ]]

### 7.3. 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 [I-D.ietf-oauth-discovery].

#### 7.3.1. 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

- [I-D.ietf-oauth-discovery]  
Jones, M., Sakimura, N., and J. Bradley, "OAuth 2.0 Authorization Server Metadata", draft-ietf-oauth-discovery-05 (work in progress), January 2017.
- [I-D.ietf-oauth-native-apps]  
Denniss, W. and J. Bradley, "OAuth 2.0 for Native Apps", draft-ietf-oauth-native-apps-07 (work in progress), January 2017.
- [IANA.OAuth.Parameters]  
IANA, "OAuth Parameters",  
<<http://www.iana.org/assignments/oauth-parameters>>.
- [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>>.
- [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>>.
- [RFC6755] Campbell, B. and H. Tschofenig, "An IETF URN Sub-Namespace for OAuth", RFC 6755, DOI 10.17487/RFC6755, October 2012, <<http://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, <<http://www.rfc-editor.org/info/rfc6819>>.

## Appendix A. Acknowledgements

The -00 version of this document was based on draft-recordon-oauth-v2-device edited by David Recordon and Brent Goldman. The content of that document was initially part of the OAuth 2.0 protocol specification but was later removed due to the lack of sufficient deployment expertise at that time. We would therefore also like to thank the OAuth working group for their work on the initial content of this specification through 2010.

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

Roshni Chandrashekhar, Marius Scurtescu, Breno de Medeiros, Stein Myrseth, Simon Moffatt, Brian Campbell, James Manger, and Justin Richer.

## Appendix B. Document History

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

-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](mailto:wdenniss@google.com)  
URI: <http://wdenniss.com/device-flow>

John Bradley  
Ping Identity

Email: [ve7jtb@ve7jtb.com](mailto:ve7jtb@ve7jtb.com)  
URI: <http://www.thread-safe.com/>

Michael B. Jones  
Microsoft

Email: [mbj@microsoft.com](mailto:mbj@microsoft.com)  
URI: <http://self-issued.info/>

Hannes Tschofenig  
ARM Limited  
Austria

Email: [Hannes.Tschofenig@gmx.net](mailto:Hannes.Tschofenig@gmx.net)  
URI: <http://www.tschofenig.priv.at>



OAuth Working Group  
Internet-Draft  
Intended status: Standards Track  
Expires: December 31, 2017

B. Campbell  
J. Bradley  
Ping Identity  
N. Sakimura  
Nomura Research Institute  
T. Lodderstedt  
YES Europe AG  
June 29, 2017

Mutual TLS Profile for OAuth 2.0  
draft-ietf-oauth-mtls-02

Abstract

This document describes Transport Layer Security (TLS) mutual authentication using X.509 certificates as a mechanism for both OAuth client authentication to the token endpoint as well as for sender constrained access to OAuth protected resources.

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 31, 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. Mutual TLS for Client Authentication . . . . .	3
2.1. Mutual TLS Client Authentication to the Token Endpoint . . . . .	3
2.2. Authorization Server Metadata . . . . .	4
2.3. Dynamic Client Registration . . . . .	4
3. Mutual TLS Sender Constrained Resources Access . . . . .	5
3.1. X.509 Certificate SHA-256 Thumbprint Confirmation Method for JWT . . . . .	5
3.2. Confirmation Method for Token Introspection . . . . .	6
4. IANA Considerations . . . . .	7
4.1. JWT Confirmation Methods Registration . . . . .	7
4.1.1. Registry Contents . . . . .	7
4.2. Token Endpoint Authentication Method Registration . . . . .	7
4.2.1. Registry Contents . . . . .	8
4.3. OAuth Token Introspection Response Registration . . . . .	8
4.3.1. Registry Contents . . . . .	8
4.4. OAuth Dynamic Client Registration Metadata Registration . . . . .	8
4.4.1. Registry Contents . . . . .	8
5. Security Considerations . . . . .	8
5.1. TLS Versions and Best Practices . . . . .	8
5.2. Client Identity Binding by the Authorization Server . . . . .	9
6. References . . . . .	9
6.1. Normative References . . . . .	9
6.2. Informative References . . . . .	10
Appendix A. Acknowledgements . . . . .	11
Appendix B. Document(s) History . . . . .	11
Authors' Addresses . . . . .	12

## 1. Introduction

This document describes Transport Layer Security (TLS) mutual authentication using X.509 certificates as a mechanism for both OAuth client authentication to the token endpoint as well as for sender constrained access to OAuth protected resources.

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, which provides better security characteristics than shared secrets.

Mutual TLS sender constrained access to protected resources ensures that only the party in possession of the private key corresponding to the certificate can utilize the access token to get access to the associated resources. Such a constraint is unlike the case of the basic bearer token described in [RFC6750], where any party in possession of the access token can use it to access the associated resources. Mutual TLS sender constrained access binds the access token to the client's certificate thus preventing the use of stolen access tokens or replay of access tokens by unauthorized parties.

Mutual TLS sender constrained access tokens and mutual TLS client authentication are distinct mechanisms that don't necessarily need to be deployed together.

### 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 following phrases interchangeably:

Transport Layer Security (TLS) Mutual Authentication

Mutual TLS

These phrases all refer to the process whereby a client uses its X.509 certificate to authenticate itself with a server when negotiating a TLS session. In TLS 1.2 [RFC5246] this requires the client to send Client Certificate and Certificate Verify messages during the TLS handshake and for the server to verify these messages.

## 2. Mutual TLS for Client Authentication

### 2.1. Mutual TLS Client Authentication to the Token Endpoint

The following section defines, as an extension of OAuth 2.0, Section 2.3 [RFC6749], the use of mutual TLS X.509 client certificates as client credentials. The requirement of mutual TLS for client authentications is determined by the authorization server based on policy or configuration for the given client (regardless of

whether the client was dynamically registered or statically configured or otherwise established). 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. As described in Section 5.2, the authorization server MUST enforce some method of binding a certificate to a client.

## 2.2. Authorization Server Metadata

"tls\_client\_auth" is used as a new value of the "token\_endpoint\_auth\_methods\_supported" metadata parameter to 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].

## 2.3. Dynamic Client Registration

This draft adds the following values and metadata parameters to OAuth 2.0 Dynamic Client Registration [RFC7591].

The value "tls\_client\_auth" is used to indicate the client's intention to use mutual TLS as an authentication method to the token endpoint for the "token\_endpoint\_auth\_method" client metadata field.

For authorization servers that associate certificates with clients using subject information in the certificate, the following two new metadata parameters can be used:

tls\_client\_auth\_subject\_dn

An [RFC4514] string representation of the expected subject distinguished name of the certificate the OAuth client will use in mutual TLS authentication.

tls\_client\_auth\_root\_dn

An [RFC4514] string representation of a distinguished name that can optionally be used to constrain, for the given client, the expected distinguished name of the root issuer of the client certificate.

For authorization servers that use the key or full certificate to associate clients with certificates, the existing "jwks\_uri" or "jwks" metadata parameters from [RFC7591] should be used.

### 3. Mutual TLS Sender Constrained Resources Access

When mutual TLS is used at the token endpoint, the authorization server is able to bind the issued access token to the client certificate. Such a binding is accomplished by associating the certificate with the token in a way that can be accessed by the protected resource, such as embedding the certificate hash in the issued access token directly, using the syntax described in Section 3.1, or through token introspection as described in Section 3.2. Other methods of associating a certificate with an access token are possible, per agreement by the authorization server and the protected resource, but are beyond the scope of this specification.

The client makes protected resource requests as described in [RFC6750], however, those requests MUST be made over a mutually authenticated TLS connection using the same certificate that was used for mutual TLS at the token endpoint.

The protected resource MUST obtain the client certificate used for mutual TLS authentication and MUST verify that the certificate matches the certificate associated with the access token. If they do not match, the resource access attempt MUST be rejected with an error.

#### 3.1. X.509 Certificate SHA-256 Thumbprint Confirmation Method for JWT

When access tokens are represented as a JSON Web Tokens (JWT)[RFC7519], the certificate hash information SHOULD be represented using the "x5t#S256" confirmation method member defined herein.

To represent the hash of a certificate in a JWT, this specification defines the new JWT Confirmation Method RFC 7800 [RFC7800] member "x5t#S256" for the X.509 Certificate SHA-256 Thumbprint. The value of the "x5t#S256" member is a base64url-encoded SHA-256[SHS] hash (a.k.a. thumbprint or digest) of the DER encoding of the X.509 certificate[RFC5280] (note that certificate thumbprints are also sometimes also known as certificate fingerprints).

The following is an example of a JWT payload containing an "x5t#S256" certificate thumbprint confirmation method.

```
{
  "iss": "https://server.example.com",
  "sub": "ty.webb@example.com",
  "exp": 1493726400,
  "nbf": 1493722800,
  "cnf": {
    "x5t#S256": "bwcK0esc3ACC3DB2Y5_lESsXE8o9ltc05O89jdN-dg2"
  }
}
```

Figure 1: Example claims of a Certificate Thumbprint Constrained JWT

### 3.2. Confirmation Method for Token Introspection

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 mutual TLS sender constrained access token, the hash of the certificate 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 certificate SHA-256 thumbprint confirmation method, described in Section 3.1, as a top-level member of the introspection response JSON. The protected resource compares that certificate hash to a hash of the client certificate used for mutual TLS authentication and rejects the request, if they do not match.

Proof-of-Possession Key Semantics for JSON Web Tokens [RFC7800] defined the "cnf" (confirmation) claim, which enables confirmation key information to be carried in a JWT. However, the same proof-of-possession semantics are also useful for introspected access tokens whereby the protected resource obtains the confirmation key data as meta-information of a token introspection response and uses that information in verifying proof-of-possession. Therefore this specification defines and registers proof-of-possession semantics for OAuth 2.0 Token Introspection [RFC7662] using the "cnf" structure. When included as a top-level member of an OAuth token introspection response, "cnf" has the same semantics and format as the claim of the same name defined in [RFC7800]. While this specification only explicitly uses the "x5t#S256" confirmation method member, it needed to define and register the higher level "cnf" structure as an introspection response member in order to define and use its more specific "x5t#S256" confirmation method.

The following is an example of an introspection response for an active token with an "x5t#S256" certificate thumbprint confirmation method.

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

{
  "active": true,
  "iss": "https://server.example.com",
  "sub": "ty.webb@example.com",
  "exp": 1493726400,
  "nbf": 1493722800,
  "cnf": {
    "x5t#S256": "bwcK0esc3ACC3DB2Y5_lESsXE8o9ltc05O89jdN-dg2"
  }
}
```

Figure 2: Example Introspection Response for a Certificate Constrained Access Token

## 4. IANA Considerations

### 4.1. JWT Confirmation Methods Registration

This specification requests registration of the following value in the IANA "JWT Confirmation Methods" registry [IANA.JWT.Claims] for JWT "cnf" member values established by [RFC7800].

#### 4.1.1. Registry Contents

- o Confirmation Method Value: "x5t#S256"
- o Confirmation Method Description: X.509 Certificate SHA-256 Thumbprint
- o Change Controller: IESG
- o Specification Document(s): Section 3.1 of [[ this specification ]]

### 4.2. Token Endpoint Authentication Method Registration

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

#### 4.2.1. Registry Contents

- o Token Endpoint Authentication Method Name: "tls\_client\_auth"
- o Change Controller: IESG
- o Specification Document(s): Section 2.2 of [[ this specification ]]

#### 4.3. OAuth Token Introspection Response Registration

This specification requests registration of the following value in the IANA "OAuth Token Introspection Response" registry [IANA.OAuth.Parameters] established by [RFC7662].

##### 4.3.1. Registry Contents

- o Claim Name: "cnf"
- o Claim Description: Confirmation
- o Change Controller: IESG
- o Specification Document(s): Section 3.2 of [[ this specification ]]

#### 4.4. OAuth Dynamic Client Registration Metadata Registration

This specification requests registration of the following client metadata definitions in the IANA "OAuth Dynamic Client Registration Metadata" registry [IANA.OAuth.Parameters] established by [RFC7591]:

##### 4.4.1. Registry Contents

- o Client Metadata Name: "tls\_client\_auth\_subject\_dn"
- o Client Metadata Description: String value specifying the expected subject distinguished name of the client certificate.
- o Change Controller: IESG
- o Specification Document(s): Section 2.3 of [[ this specification ]]
  
- o Client Metadata Name: "tls\_client\_auth\_root\_dn"
- o Client Metadata Description: String value specifying the expected distinguished name of the root issuer of the client certificate
- o Change Controller: IESG
- o Specification Document(s): Section 2.3 of [[ 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 by the Authorization Server

No specific method of binding a certificate to a client identifier at the token endpoint is prescribed by this document. However, some method **MUST** 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 and possibly the expected trust anchor. 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>>.
- [RFC4514] Zeilenga, K., Ed., "Lightweight Directory Access Protocol (LDAP): String Representation of Distinguished Names", RFC 4514, DOI 10.17487/RFC4514, June 2006, <<http://www.rfc-editor.org/info/rfc4514>>.
- [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>>.
- [RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <<http://www.rfc-editor.org/info/rfc5280>>.

- [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>>.
- [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, <<http://www.rfc-editor.org/info/rfc7800>>.
- [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>>.

## 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.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.Discovery] Sakimura, N., Bradley, J., Jones, M., and E. Jay, "OpenID Connect Discovery 1.0", February 2014.
- [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>>.
- [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>>.

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

## Appendix A. Acknowledgements

Scott "not Tomlinson" Tomilson and Matt Peterson were involved in design and development work on a mutual TLS OAuth client authentication implementation that informed some of the content of this document.

Additionally, the authors would like to thank the following people for their input and contributions to the specification: Sergey Beryozkin, Vladimir Dzhuvinov, Samuel Erdtman, Phil Hunt, Sean Leonard, Kepeng Li, James Manger, Jim Manico, Nov Mataka, Sascha Preibisch, Justin Richer, Dave Tonge, and Hannes Tschofenig.

## Appendix B. Document(s) History

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

draft-ietf-oauth-mtls-02

- o Fixed editorial issue <https://mailarchive.ietf.org/arch/msg/oauth/U46UMeh8XIOQnvXY9pHFqlMKPns>
- o Changed the title (hopefully "Mutual TLS Profile for OAuth 2.0" is better than "Mutual TLS Profiles for OAuth Clients").

draft-ietf-oauth-mtls-01

- o Added more explicit details of using RFC 7662 token introspection with mutual TLS sender constrained access tokens.
- o Added an IANA OAuth Token Introspection Response Registration request for "cnf".
- o Specify that `tls_client_auth_subject_dn` and `tls_client_auth_root_dn` are RFC 4514 String Representation of Distinguished Names.
- o Changed `tls_client_auth_issuer_dn` to `tls_client_auth_root_dn`.
- o Changed the text in the Section 3 to not be specific about using a hash of the cert.
- o Changed the abbreviated title to 'OAuth Mutual TLS' (previously was the acronym MTLSPOC).

draft-ietf-oauth-mtls-00

- o Created the initial working group version from draft-campbell-oauth-mtls

draft-campbell-oauth-mtls-01

- o Fix some typos.
- o Add to the acknowledgements list.

draft-campbell-oauth-mtls-00

- o Add a Mutual TLS sender constrained protected resource access method and a x5t#S256 cnf method for JWT access tokens (concepts taken in part from draft-sakimura-oauth-jpop-04).
- o Fixed "token\_endpoint\_auth\_methods\_supported" to "token\_endpoint\_auth\_method" for client metadata.
- o Add "tls\_client\_auth\_subject\_dn" and "tls\_client\_auth\_issuer\_dn" client metadata parameters and mention using "jwks\_uri" or "jwks".
- o Say that the authentication method is determined by client policy regardless of whether the client was dynamically registered or statically configured.
- o Expand acknowledgements to those that participated in discussions around draft-campbell-oauth-tls-client-auth-00
- o Add Nat Sakimura and Torsten Lodderstedt to the author list.

draft-campbell-oauth-tls-client-auth-00

- o Initial draft.

#### Authors' Addresses

Brian Campbell  
Ping Identity

Email: [brian.d.campbell@gmail.com](mailto:brian.d.campbell@gmail.com)

John Bradley  
Ping Identity

Email: [ve7jtb@ve7jtb.com](mailto:ve7jtb@ve7jtb.com)  
URI: <http://www.thread-safe.com/>

Nat Sakimura  
Nomura Research Institute

Email: [n-sakimura@nri.co.jp](mailto:n-sakimura@nri.co.jp)  
URI: <https://nat.sakimura.org/>

Torsten Lodderstedt  
YES Europe AG

Email: [torsten@lodderstedt.net](mailto:torsten@lodderstedt.net)

Open Authentication Protocol  
Internet-Draft  
Intended status: Best Current Practice  
Expires: September 29, 2017

T. Lodderstedt, Ed.  
YES Europe AG  
J. Bradley  
Ping Identity  
A. Labunets  
Facebook  
March 30, 2017

OAuth Security Topics  
draft-ietf-oauth-security-topics-02

Abstract

This draft gives a comprehensive overview on open OAuth security topics. It is intended to serve as a working document for the OAuth working group to systematically capture and discuss these security topics and respective mitigations and eventually recommend best current practice and also OAuth extensions needed to cope with the respective security threats.

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 29, 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. Recommended Best Practice . . . . .	3
2.1. Protecting redirect-based flows . . . . .	4
2.2. TBD . . . . .	4
3. Recommended modifications and extensions to OAuth . . . . .	4
4. OAuth Credentials Leakage . . . . .	5
4.1. Insufficient redirect URI validation . . . . .	5
4.1.1. Attacks on Authorization Code Grant . . . . .	5
4.1.2. Attacks on Implicit Grant . . . . .	6
4.1.3. Proposed Countermeasures . . . . .	7
4.2. Authorization code leakage via referrer headers . . . . .	9
4.2.1. Proposed Countermeasures . . . . .	9
4.3. Attacks in the Browser . . . . .	9
4.3.1. Code in browser history (TBD) . . . . .	9
4.3.2. Access token in browser history (TBD) . . . . .	10
4.3.3. Javascript Code stealing Access Tokens (TBD) . . . . .	10
4.4. Dynamic OAuth Scenarios . . . . .	10
4.4.1. Access Token Phishing by Counterfeit Resource Server . . . . .	10
4.4.2. Mix-Up . . . . .	11
5. OAuth Credentials Injection . . . . .	12
5.1. Code Injection . . . . .	12
5.1.1. Proposed Countermeasures . . . . .	14
5.1.2. Access Token Injection (TBD) . . . . .	15
5.1.3. XSRF (TBD) . . . . .	16
6. Other Attacks . . . . .	16
7. Other Topics . . . . .	16
8. Acknowledgements . . . . .	16
9. IANA Considerations . . . . .	16
10. Security Considerations . . . . .	16
11. References . . . . .	16
11.1. Normative References . . . . .	17
11.2. Informative References . . . . .	17
Appendix A. Document History . . . . .	17
Authors' Addresses . . . . .	18

## 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. While OAuth was used in a variety of scenarios and different kinds of deployments, the following challenges could be observed:

- 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 RFC 6749 [RFC6749], RFC 6750 [RFC6749], and RFC 6819 [RFC6819].

The remainder of the document is organized as follows: The next section gives a summary of the set of security mechanisms and practices, the working group shall consider to recommend to OAuth implementers. This is followed by a section proposing modifications to OAuth intended to either simplify its usage and to strengthen its security.

The remainder of the draft gives a detailed analyses of the weaknesses and implementation issues, which can be found in the wild today along with a discussion of potential counter measures. First, various scenarios how OAuth credentials (namely access tokens and authorization codes) may be disclosed to attackers and proposes countermeasures are discussed. Afterwards, the document discusses attacks possible with captured credential and how they may be prevented. The last sections discuss additional threats.

## 2. Recommended Best Practice

This section describes the set of security mechanisms the authors



believe should be taken into consideration by the OAuth working group to be recommended to OAuth implementers.

## 2.1. Protecting redirect-based flows

Authorization servers shall utilize exact matching of client redirect URIs against pre-registered URIs. This measure contributes to the prevention of leakage of authorization codes and access tokens (depending on the grant type). It also helps to detect mix up attacks.

Clients shall avoid any redirects or forwards, which can be parameterized by URI query parameters, in order to provide a further layer of defence against token leakage. If there is a need for this kind of redirects, clients are advised to implement appropriate counter measures against open redirection, e.g. as described by the OWASP [owasp].

Clients shall ensure to only process redirect responses of the OAuth authorization server they send the respective request to and in the same user agent this request was initiated in. In particular, clients shall implement appropriate XSRF prevention by utilizing one-time use XSRF tokens carried in the STATE parameter, which are securely bound to the user agent. Moreover, the client shall store the authorization server's identity it send an authorization request to in a transaction-specific manner, which is also bound to the particular user agent. Furthermore, clients should use AS-specific redirect URIs as a means to identify the AS a particular response came from. Matching this with the before mentioned information regarding the AS the client sent the request to helps to detect mix-up attacks.

Note: [I-D.bradley-oauth-jwt-encoded-state] gives advice on how to implement XSRF prevention and AS matching using signed JWTs in the STATE parameter.

Clients shall use PKCE [RFC7636] in order to (with the help of the authorization server) detect attempts to inject authorization codes into the authorization response. The PKCE challenges must be transaction-specific and securely bound to the user agent, in which the transaction was started.

Note: although PKCE so far was recommended as mechanism to protect native apps, this advice applies to all kinds of OAuth clients, including web applications.

## 2.2. TBD

## 3. Recommended modifications and extensions to OAuth

This section describes the set of modifications and extensions the authors believe should be taken into consideration by the OAuth working group change and extend OAuth in order to strengthen its security and make it simpler to implement. It also recommends some changes to the OAuth set of specs.

Remove requirement to check actual redirect URI at token endpoint - seems to be complicated to implement properly and could be compromised

#### 4. OAuth Credentials Leakage

This section describes a couple of different ways how OAuth credentials, namely authorization codes and access tokens, can be exposed to attackers.

##### 4.1. Insufficient redirect URI validation

Some authorization servers allow clients to register redirect URI patterns instead of complete redirect URIs. In those cases, the authorization server, at runtime, matches 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 by exposing the OAuth credentials to an attacker by utilizing an open redirector at the client in conjunction with the way user agents handle URL fragments.

##### 4.1.1. Attacks on Authorization Code Grant

For a public client using the grant type code, an attack would look as follows:

Let's assume the redirect URL pattern "https://\*.example.com/\*" had been registered for the client "s6BhdRkqt3". This pattern allows redirect URIs 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
```

- (4 )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.
- (5 )If the user does not recognize the attack, the code is issued and directly sent to the attacker's client.
- (6 )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 directly 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 (e.g. by mounting a code injection attack). This and other kinds of injections are covered in Section OAuth Credentials Injection.

#### 4.1.2. Attacks on Implicit Grant

The attack described above works for the implicit grant as well. 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, implicit clients can be subject to a further kind of attacks. It utilizes the fact that 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). The attack described here combines this behavior with the client as an open redirector in order to get access to access tokens. This allows circumvention even of strict redirect URI patterns (but not strict URL matching!).

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" (line breaks 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
```

- (5 )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&...
```

- (6 )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
```

- (7 )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&...
```

- (8 )The attacker's page at client.evil.com can access the fragment and obtain the access token.

#### 4.1.1.3. Proposed Countermeasures

The complexity of implementing and managing pattern matching correctly obviously causes security issues. This document therefore proposes to simplify the required logic and configuration by using exact redirect URI matching only. This means 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 on whether this feature is really used in deployments today.
- o The working group may also consider this change as a step towards improved interoperability for OAuth implementations since RFC 6749 is somewhat vague on redirect URI validation. Notably there are no rules for pattern matching. One may therefore 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 and foster security?

- 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.

Question: Does redirect uri validation solve any problem for native apps? Effective against impersonation when used in conjunction with claimed HTTPS redirect URIs only.

For Windows token broker exact redirect URI matching is important as the redirect URI encodes the app identity. For custom scheme redirects there is a question however it is probably a useful part of defense in depth.

Additional recommendations:

- o Servers on which callbacks are hosted must not expose open redirectors (see respective section).
- o Clients may drop fragments via intermediary URLs with "fix fragments" (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:

- o authenticate client using digital signatures (JAR? <https://tools.ietf.org/html/draft-ietf-oauth-jwsreq-09>)

#### 4.2. Authorization code leakage via referrer headers

It is 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 <img> (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.

##### 4.2.1. Proposed 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 (don't transport credentials via URL parameters and GET)

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>)

#### 4.3. Attacks in the Browser

#### 4.3.1. 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.

Proposed countermeasures: code is one time use, has limited duration, is bound to client id/secret (confidential clients only)

#### 4.3.2. 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

#### 4.3.3. Javascript Code stealing Access Tokens (TBD)

sandboxing using service workers

#### 4.4. Dynamic OAuth Scenarios

OAuth initially assumed a static relationship between client, authorization server and resource servers. The URLs of AS and RS were known to the client at deployment time and built an anchor for the trust relationship among those parties. The validation whether the client talks to a legitimate server is based on TLS server authentication (see [RFC6819], Section 4.5.4).

With the increasing adoption of OAuth, this simple model dissolved and, in several scenarios, was replaced by a dynamic establishment of the relationship between clients on one side and the authorization and resource servers of a particular deployment on the other side. This way the same client can be used to access services of different providers (in case of standard APIs, such as e-Mail or OpenID Connect) or serves as a frontend to a particular tenant in a multi-tenancy.

Extensions of OAuth, such as [RFC7591] and [I-D.ietf-oauth-discovery] were developed in order to support the usage of OAuth in dynamic scenarios.

As a challenge to the community, such usage scenarios open up new attack angles, which are discussed in this section.

##### 4.4.1. Access Token Phishing by Counterfeit Resource Server

An attacker may pretend to be a particular resource server and to accept tokens from a particular authorization server. If the client sends a valid access token to this counterfeit resource server, the server in turn may use that token to access other services on behalf of the resource owner.

Potential mitigation strategies:

- o AS may publish information about its legitimate resource servers, clients must only send access tokens to this servers
- o Clients indicate resource server they intend to use the access token for at AS, AS may refuse to issue tokens for resource servers it does not know
- o AS indicates resource servers a particular access token is good for to client - client enforced audience restriction - prevents disclosure (e.g. OAuth Response Metadata (<https://tools.ietf.org/html/draft-sakimura-oauth-meta-07>))
- o Access tokens are audience restricted - prevents replay if the audience is a URL determined by the client, reduces impact in case of legitimate resource server uses token at other resource server (e.g. <https://tools.ietf.org/html/draft-campbell-oauth-resource-indicators-01>)
- o Access Token is sender restricted - sender is cryptographically verified
  - \* <https://tools.ietf.org/html/draft-ietf-oauth-pop-architecture-08>
  - \* <https://tools.ietf.org/html/draft-jones-oauth-token-binding-00>
  - \* <https://datatracker.ietf.org/doc/draft-campbell-oauth-mtls>
  - \* <https://datatracker.ietf.org/doc/html/draft-sakimura-oauth-jpop>

#### 4.4.2. Mix-Up

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 countermeasures 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 carries client id and issuer (requires OpenID Connect)
- o Clients use AS-specific redirect URIs, for every authorization request store intended AS and compare intention with actual redirect URI where the response was received (no change to OAuth required)

## 5. 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).

### 5.1. 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 is 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.

#### 5.1.1. Proposed Countermeasures

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

Code-bound 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.

Note: token binding could be used in conjunction with PKCE as an option (<https://tools.ietf.org/html/draft-campbell-oauth-tbpkce>).

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 countermeasures described above if he is able to create or capture the respective secret or code\_challenge 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 and 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)?

#### 5.1.2. 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

#### 5.1.3. 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))

#### 6. 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

#### 7. Other Topics

why to rotate refresh tokens

why audience restriction

how to support multi AS per RS

...

differentiate native, JS and web clients

federated login to apps (code flow to own AS in browser and federated  
login to 3rd party IDP in browser)

do not put sensitive data in URL/GET parameters (Jim Manico)

#### 8. Acknowledgements

We would like to thank Jim Manico and Phil Hunt for their valuable  
feedback.

#### 9. IANA Considerations

This draft includes no request to IANA.

#### 10. Security Considerations

All relevant security considerations have been given in the  
functional specification.

#### 11. References

## 11.1. 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>>.
- [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>>.

## 11.2. Informative References

- [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", Internet-Draft draft-bradley-oauth-jwt-encoded-state-07, March 2017.
- [I-D.ietf-oauth-discovery]  
Jones, M., Sakimura, N. and J. Bradley, "OAuth 2.0 Authorization Server Metadata", Internet-Draft draft-ietf-oauth-discovery-04, August 2016.
- [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>>.
- [owasp] "Open Web Application Security Project Home Page", , <<https://www.owasp.org/>>.

## Appendix A. Document History

[[ To be removed from the final specification ]]

-01

- o Added references to mitigation methods for token leakage
- o Added reference to Token Binding for Authorization Code
- o incorporated feedback of Phil Hunt
- o fixed numbering issue in attack descriptions in section 2

-00 (WG document)

- o turned the ID into a WG document and a BCP
- o Added federated app login as topic in Other Topics

#### Authors' Addresses

Torsten Lodderstedt, editor  
YES Europe AG

Email: [torsten@lodderstedt.net](mailto:torsten@lodderstedt.net)

John Bradley  
Ping Identity

Email: [ve7jtb@ve7jtb.com](mailto:ve7jtb@ve7jtb.com)

Andrey Labunets  
Facebook

Email: [isciurus@fb.com](mailto:isciurus@fb.com)

OAuth Working Group  
Internet-Draft  
Intended status: Standards Track  
Expires: January 4, 2018

M. Jones  
Microsoft  
J. Bradley  
B. Campbell  
Ping Identity  
W. Denniss  
Google  
July 3, 2017

OAuth 2.0 Token Binding  
draft-ietf-oauth-token-binding-04

Abstract

This specification enables OAuth 2.0 implementations to apply Token Binding to Access Tokens, Authorization Codes, and Refresh Tokens. 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 <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 4, 2018.

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 . . . . .	3
1.1. Requirements Notation and Conventions . . . . .	3
1.2. Terminology . . . . .	3
2. Token Binding for Refresh Tokens . . . . .	3
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 for Authorization Codes . . . . .	13
4.1. Native Application Clients . . . . .	13
4.1.1. Code Challenge . . . . .	14
4.1.1.1. Example Code Challenge . . . . .	14
4.1.2. Code Verifier . . . . .	14
4.1.2.1. Example Code Verifier . . . . .	15
4.2. Web Server Clients . . . . .	15
4.2.1. Code Challenge . . . . .	16
4.2.1.1. Example Code Challenge . . . . .	16
4.2.2. Code Verifier . . . . .	17
4.2.2.1. Example Code Verifier . . . . .	18
5. Phasing in Token Binding and Preventing Downgrade Attacks . .	18
6. Token Binding Metadata . . . . .	19
6.1. Token Binding Client Metadata . . . . .	19
6.2. Token Binding Authorization Server Metadata . . . . .	20
6.3. Token Binding Protected Resource Metadata . . . . .	20
7. Security Considerations . . . . .	20
8. IANA Considerations . . . . .	20
8.1. OAuth Dynamic Client Registration Metadata Registration .	20
8.1.1. Registry Contents . . . . .	21
8.2. OAuth Authorization Server Metadata Registration . . . .	21
8.2.1. Registry Contents . . . . .	21
8.3. OAuth Protected Resource Metadata Registration . . . . .	21
8.3.1. Registry Contents . . . . .	21

8.4. PKCE Code Challenge Method Registration . . . . .	22
8.4.1. Registry Contents . . . . .	22
9. References . . . . .	22
9.1. Normative References . . . . .	22
9.2. Informative References . . . . .	24
Appendix A. Acknowledgements . . . . .	24
Appendix B. Open Issues . . . . .	24
Appendix C. Document History . . . . .	25
Authors' Addresses . . . . .	27

## 1. Introduction

This specification enables OAuth 2.0 [RFC6749] implementations to apply Token Binding (TLS Extension for Token Binding Protocol Negotiation [I-D.ietf-tokbind-negotiation], The Token Binding Protocol Version 1.0 [I-D.ietf-tokbind-protocol] and Token Binding over HTTP [I-D.ietf-tokbind-https]) to Access Tokens, Authorization Codes, and Refresh Tokens. 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 RFC 2119 [RFC2119].

### 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 [I-D.ietf-tokbind-https].

## 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 [I-D.ietf-tokbind-https]. 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 federated scenarios described in Section 4 (Federation Use Cases) of Token Binding over HTTP [I-D.ietf-tokbind-https] 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 [I-D.ietf-oauth-native-apps]. 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: AIAAgBBQGto7hHRR0Y5nkOWqc9KNfwW95dEFmSI_tCZ_Cbl
  7LWlt6Xjp3DbjiDjavGFikP2HV_2JSE42VzmKOVVV8m7eqAAQOKiDK10i0z6v4X5B
  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: AIAAgBBQGto7hHRR0Y5nkOWqc9KNfwW95dEFmSI_tCZ_Cbl
  7LWlt6Xjp3DbjiDJavGFikP2HV_2JSE42VzmKOVVV8m7eqAAQCpGbaG_YRf27qOra
  L0UT4fsKKjL6PukuOT00qzamoAXxOq7m_id7O3mLpnb_sM7kwSxLi7iNHzzDgCAkP
  t3lHwAAAA

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 Section 4 (Federation Use Cases) of Token Binding over HTTP [I-D.ietf-tokbind-https]. 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 Section 5 (Implementation Considerations) of Token Binding over HTTP [I-D.ietf-tokbind-https].

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 Section 4.2 of OAuth 2.0 [RFC6749], 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.

### 3.1.1.1. 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=rM8pZxG1c3gKy6rEbsD8s
  &redirect_uri=https%3A%2F%2Fclient%2Eexample%2Eorg%2Fcb HTTP/1.1
Host: server.example.com
Sec-Token-Binding: ARIAAGBBQIEE8mSMtDy2dj9EEBdXaQT9W3Rq1NS-jW8ebPoF
6FyL0jIfATVE55zlircgOTZmEglxeIrc3DsGegwjs4bhw14AQGKDLAXFFMyQkZegC
wlbTlqX3F9HTt-1JxFU_pil6ezka7qVRCpSF0BQLfSqlsxMbYfSSCJX1BDtrIL7PX
j__fUAAAECAEFAlBNUnP3te5WrwlEwiejEz0OpesmC5PElWc7kZ5nlLSqQTj1ciIp
5vQ30LLUCyM_a2BYTUPKtd5EdS-PalT4t6ABADgeizRa5NkTMuX4zOdC-R4cLNWVV
08lLu2Psko-UJLR_XAH4Q0H7-m0_nQR1zBN78nYMKPvHsz8L3zWKRvYXEgAA
```

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=rM8pZxG1c3gKy6rEbsD8s
  &expires_in=3600&token_type=Bearer
  &access_token=eyJhbGciOiJIUzI1IiwiaXNjaWkiOiJ0eXN5W5sQ
```

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_NmNCl9JXy4NkdkeZBxqvZGPr0y8QZ_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": "eyJhbGciOiJIJFZlIiwiaXNjaW50IjoiImtp[...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.

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": "7NRBu9iDdJlYCTOqyeYuLxXv0blEA-yTpmGlrAwKAws"
  }
}
```

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 an "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-yTpmGlrAwKAws"
  }
}
```

Figure 13: Example Introspection Response for a Token Bound Access Token

#### 4. 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.

##### 4.1. 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.

#### 4.1.1.1. 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.

##### 4.1.1.1.1. 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

#### 4.1.1.2. Code Verifier

Upon receipt of the authorization code, the client sends the access token request to the token endpoint. The Token Binding Protocol [I-D.ietf-tokbind-protocol] 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 [I-D.ietf-tokbind-https], 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.

#### 4.1.2.1. 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: AIkAAgBBQEOO9GRFP-LM0hoWw6-2i3l8BsuuUum5AL8bt1sz
  1rlEFfp5DMXMNW3O8WjcIXr2DKJnI4xnuGse6GywQd9RbD0AQJDb3xyo9PBxj8M6Y
  jLt-6OaxgDkyoBoTKyrrNbLc8tJQ0JtXomKzBbj5qPtHDduXc6xz_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

#### 4.2. 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.

#### 4.2.1. 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 Section 5.3 of Token Binding over HTTP [I-D.ietf-tokbind-https].

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.

##### 4.2.1.1. 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=P4FUFqYzslij3ffsYCP34d3
        &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=dry08YFpWacBUPjhBf4Nvt51
    &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_OVfBNqrIQxzIfkF7tWby2ZfyaE6XpwTsAQBYqhFX78vM0gDX_F
    d_b2dlHyHlMmkIz8iMVB_Y_reM98OUaJFz5IB7PG9nZ11j58LoG5QhmQoI9NXYktKZ
    RXxrYAAAECAEFAdUFTnfQADknluDbQnvJEk6oQs38L92gv-KO-qlYadLoDIKe2h53
    hSiKwIP98iRj_unedkNkAMyg9e2mY4Gp7WwBAeDUOwaSXNzle6gKohwN4SAZ5eNyx
    45Mh8VI4woLlBipLoqrJRoK6dxFkWgHRMuBROcLGUj5PiOoxybQH_Tom3gAA

```

Figure 17: Authorization Request

#### 4.2.2. 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.



#### 4.2.2.1. 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=dry08YFpWacBUPjhBf4Nvt5l&code=jwD3oOa5cQvvLc8lBwc4CMw
Host: client.example.org
Sec-Token-Binding: AIkAAgBBQHVBu530AA5J9bg20J7yRJOqELN_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=jwD3oOa5cQvvLc8lBwc4CMw
&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

### 5. Phasing in Token Binding and Preventing Downgrade Attacks

Many OAuth implementations will be deployed in situations in which not all participants support Token Binding. Any of 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 Token Binding failures at any step as fatal errors. 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 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.

If all the participants in an authorization interaction support Token Binding and yet one or more of them does not use it, this is likely evidence of a downgrade attack. In this case, the authorization SHOULD be aborted with an error. For instance, if the protected resource knows that the authorization server and the user agent both support Token Binding and yet the access token received does not contain Token Binding information, this is almost certainly a sign of an attack.

The authorization server, client, and protected resource can determine whether the others support Token Binding using the metadata values defined in the next section. They can determine whether the user agent supports Token Binding by whether it negotiated Token Binding for the TLS connection.

## 6. Token Binding Metadata

### 6.1. 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".

## 6.2. Token Binding Authorization Server Metadata

Authorization servers supporting Token Binding that also support OAuth 2.0 Authorization Server Metadata [OAuth.AuthorizationMetadata] 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".

## 6.3. Token Binding Protected Resource Metadata

Protected resources supporting Token Binding that also support the OAuth 2.0 Protected Resource Metadata [OAuth.ResourceMetadata] use this metadata value to declare their support for Token Binding of access tokens:

`resource_access_token_token_binding_supported`

OPTIONAL. Boolean value specifying whether the protected resource supports Token Binding of access tokens. If omitted, the default value is "false".

## 7. Security Considerations

If a refresh request is received by the authorization server containing a Referred Token Binding ID and the refresh token in the request is not itself token bound, then it is not clear that token binding the access token adds significant value. This situation should be considered an open issue for discussion by the working group.

## 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 6.1 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 6.1 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 established by [OAuth.AuthorizationMetadata]:

##### 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 6.2 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 6.2 of [[ this specification ]]

#### 8.3. OAuth Protected Resource Metadata Registration

This specification registers the following client metadata definition in the IANA "OAuth Protected Resource Metadata" registry established by [OAuth.ResourceMetadata]:

##### 8.3.1. Registry Contents

- o Resource Metadata Name:  
"resource\_access\_token\_token\_binding\_supported"
- o Resource Metadata Description: Boolean value specifying whether the protected resource supports Token Binding of access tokens

- o Change Controller: IESG
- o Specification Document(s): Section 6.3 of [[ this specification ]]

#### 8.4. 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].

##### 8.4.1. Registry Contents

- o Code Challenge Method Parameter Name: TB-S256
- o Change controller: IESG
- o Specification document(s): Section 4.1.1 of [[ this specification ]]
- o Code Challenge Method Parameter Name: referred\_tb
- o Change controller: IESG
- o Specification document(s): Section 4.2.1 of [[ this specification ]]

#### 9. References

##### 9.1. Normative References

[I-D.ietf-tokbind-https]

Popov, A., Nystrom, M., Balfanz, D., Langley, A., and J. Hodges, "Token Binding over HTTP", draft-ietf-tokbind-https-08 (work in progress), February 2017.

[I-D.ietf-tokbind-negotiation]

Popov, A., Nystrom, M., Balfanz, D., and A. Langley, "Transport Layer Security (TLS) Extension for Token Binding Protocol Negotiation", draft-ietf-tokbind-negotiation-07 (work in progress), February 2017.

[I-D.ietf-tokbind-protocol]

Popov, A., Nystrom, M., Balfanz, D., Langley, A., and J. Hodges, "The Token Binding Protocol Version 1.0", draft-ietf-tokbind-protocol-13 (work in progress), February 2017.

[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.AuthorizationMetadata] Jones, M., Sakimura, N., and J. Bradley, "OAuth 2.0 Authorization Server Metadata", draft-ietf-oauth-discovery-06 (work in progress), March 2017, <<http://tools.ietf.org/html/draft-ietf-oauth-discovery-06>>.
- [OAuth.ResourceMetadata] Jones, M. and P. Hunt, "OAuth 2.0 Protected Resource Metadata", draft-jones-oauth-resource-metadata-01 (work in progress), January 2017, <<http://tools.ietf.org/html/draft-jones-oauth-resource-metadata-01>>.
- [OpenID.TokenBinding] Jones, M., Bradley, J., and B. Campbell, "OpenID Connect Token Bound Authentication 1.0", July 2016, <[http://openid.net/specs/openid-connect-token-bound-authentication-1\\_0.html](http://openid.net/specs/openid-connect-token-bound-authentication-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>>.
- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <<http://www.rfc-editor.org/info/rfc4648>>.
- [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>>.
- [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>>.

- [RFC7662] Richer, J., Ed., "OAuth 2.0 Token Introspection", RFC 7662, DOI 10.17487/RFC7662, October 2015, <<http://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, <<http://www.rfc-editor.org/info/rfc7800>>.
- [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>>.

## 9.2. Informative References

- [I-D.ietf-oauth-native-apps] Denniss, W. and J. Bradley, "OAuth 2.0 for Native Apps", draft-ietf-oauth-native-apps-08 (work in progress), March 2017.
- [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](http://openid.net/specs/openid-connect-core-1_0.html)>.
- [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, <<http://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, <<http://www.rfc-editor.org/info/rfc7591>>.

## Appendix A. Acknowledgements

The authors would like to thank the following people for their contributions to the specification: Dirk Balfanz, Andrei Popov, Justin Richer, and Nat Sakimura.

## Appendix B. Open Issues

- o What should we do in the case that a refresh request for a token bound access token is received when the refresh token used in the request is not token bound?

- o Currently the only way to request a token bound access token is via the referred token binding. By definition the referred token binding also comes with the provided token binding and the provided token binding is what is used to bind the refresh token. However, web server clients will typically be distributed/clustered and very likely will not want to, or be capable of, dealing with token bound refresh tokens. Such clients will have credentials established with the AS for authenticating to the token endpoint and refresh tokens are already bound to the client. So token binding the refresh tokens doesn't add much, if anything, in this case. But accessing private token binding keys in a distributed system will be cumbersome or even impossible. Tracking and properly utilizing the association of a token binding key with each individual refresh token would also be exceptionally cumbersome (whereas client credentials are for the client and decoupled from individual refresh tokens) but without some such mechanism the token binding key cannot be changed without implicitly invalidating all the bound refresh tokens the web server client has stored for that AS. It seems necessary to provide some mechanism for a client to opt-out of having refresh tokens token bound while still allowing for token binding of access tokens.
- o Should the scope of this document include standardization or guidance on token binding of JWT Client Authentication and/or Authorization Grants from RFC 7523?
- o The Metadata (Section 6) and what can and cannot be reliably inferred from it (Section 5) need additional evaluation and work. OAuth 2.0 Protected Resource Metadata [OAuth.ResourceMetadata] is no longer a going concern, but is currently referenced herein. Boolean values do not adequately convey Token Binding support, as different components may support different key parameters types. And successful negotiation likely doesn't provide the application layer info about all the supported key parameters types but rather just the one that was negotiated.

#### Appendix C. Document History

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

-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](mailto:mbj@microsoft.com)  
URI: <http://self-issued.info/>

John Bradley  
Ping Identity

Email: [ve7jtb@ve7jtb.com](mailto:ve7jtb@ve7jtb.com)  
URI: <http://www.thread-safe.com/>

Brian Campbell  
Ping Identity

Email: [brian.d.campbell@gmail.com](mailto:brian.d.campbell@gmail.com)  
URI: [https://twitter.com/\\_\\_b\\_c](https://twitter.com/__b_c)

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

Email: [wdenniss@google.com](mailto:wdenniss@google.com)  
URI: <http://wdenniss.com/>

OAuth Working Group  
Internet-Draft  
Intended status: Standards Track  
Expires: January 4, 2018

M. Jones  
A. Nadalin  
Microsoft  
B. Campbell, Ed.  
J. Bradley  
Ping Identity  
C. Mortimore  
Salesforce  
July 3, 2017

OAuth 2.0 Token Exchange  
draft-ietf-oauth-token-exchange-09

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 <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 4, 2018.

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 . . . . .	3
1.1. Delegation vs. Impersonation Semantics . . . . .	4
1.2. Requirements Notation and Conventions . . . . .	5
1.3. Terminology . . . . .	5
2. Token Exchange Request and Response . . . . .	6
2.1. Request . . . . .	6
2.1.1. Relationship Between Resource, Audience and Scope . .	8
2.2. Response . . . . .	8
2.2.1. Successful Response . . . . .	8
2.2.2. Error Response . . . . .	10
2.3. Example Token Exchange . . . . .	10
3. Token Type Identifiers . . . . .	12
4. JSON Web Token Claims and Introspection Response Parameters .	13
4.1. "act" (Actor) Claim . . . . .	13
4.2. "scp" (Scopes) Claim . . . . .	15
4.3. "cid" (Client Identifier) Claim . . . . .	16
4.4. "may_act" (May Act For) Claim . . . . .	16
5. IANA Considerations . . . . .	17
5.1. OAuth URI Registration . . . . .	17
5.1.1. Registry Contents . . . . .	17
5.2. OAuth Parameters Registration . . . . .	18
5.2.1. Registry Contents . . . . .	18
5.3. OAuth Access Token Type Registration . . . . .	19
5.3.1. Registry Contents . . . . .	19
5.4. JSON Web Token Claims Registration . . . . .	19
5.4.1. Registry Contents . . . . .	19
5.5. OAuth Token Introspection Response Registration . . . . .	20
5.5.1. Registry Contents . . . . .	20
5.6. OAuth Extensions Error Registration . . . . .	20
5.6.1. Registry Contents . . . . .	20
6. Security Considerations . . . . .	20
7. Privacy Considerations . . . . .	21
8. References . . . . .	21
8.1. Normative References . . . . .	21
8.2. Informative References . . . . .	22
Appendix A. Additional Token Exchange Examples . . . . .	22
A.1. Impersonation Token Exchange Example . . . . .	23
A.1.1. Token Exchange Request . . . . .	23
A.1.2. Subject Token Claims . . . . .	23
A.1.3. Token Exchange Response . . . . .	24
A.1.4. Issued Token Claims . . . . .	24

A.2. Delegation Token Exchange Example . . . . .	25
A.2.1. Token Exchange Request . . . . .	25
A.2.2. Subject Token Claims . . . . .	25
A.2.3. Actor Token Claims . . . . .	26
A.2.4. Token Exchange Response . . . . .	26
A.2.5. Issued Token Claims . . . . .	27
Appendix B. Acknowledgements . . . . .	28
Appendix C. Document History . . . . .	28
Authors' Addresses . . . . .	31

## 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 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 and issuing security tokens, 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 and securing access to HTTP and RESTful resources but do not provide everything necessary to facilitate token exchange interactions.

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, which 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 AS 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

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 in so far 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.

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 RFC 2119 [RFC2119].

## 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 "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 OAuth 2.0 [RFC6749].

Client authentication to the authorization server is done using the normal mechanisms provided by OAuth 2.0. Section 2.3.1 of The OAuth 2.0 Authorization Framework [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 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.

The client makes a token exchange request to the token endpoint with an extension grant type by including the following parameters using the "application/x-www-form-urlencoded" format with a character encoding of UTF-8 in the HTTP request entity-body:

#### grant\_type

REQUIRED. The value "urn:ietf:params:oauth:grant-type:token-exchange" indicates that a token exchange is being performed.

#### resource

OPTIONAL. Indicates the physical location of 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 the location 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.



**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 rather than a physical location. 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], or a URI are examples of things that might be used as "audience" parameter values. 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 physical locations.

**scope**

OPTIONAL. A list of space-delimited, case-sensitive strings 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.

**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 this 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 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 validity of the subject token or actor token have no impact on the validity of the issued token after the exchange has occurred.

#### 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

[RFC7159], 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 of 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 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 typically used with this meaning, 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 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 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 all the target services 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" and/or "error\_uri" parameters.

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 token exchange in order to trade an access token that it received in a protected resource request for a token that it will use to call to a backend service (extra line breaks and indentation in the examples are for display purposes only).

The resource server receives the following request containing an OAuth access token in the Authorization request 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

The resource server assumes the role of the client for the token exchange and the access token from the request above 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 roll 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%20
&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 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": "eyJhbGciOiJFUzI1NiIsImtpZCI6IjllciJ9.eyJhdWQiOiJodHRwczovL2JhY2t1bmQuZXhhbXBsZS5jb20iLCJpc3MiOiJodHRwczovL2FzLmV4YW1wbGUuY29tIiwiaXNjaXNDQxOTE3NTkzLCJpYXQiOiJlbnN1YiI6ImJjQGV4YW1wbGUuY29tIiwic2NwIjpbImFwaSJdfQ.MXgnpvPMo0nhcePwnQbunD2gw_pDyCFA-Saobl6gyLAdyPbaALFuAOyFc4XTWaPEHV_LGmXklSTpz0yC7hlSQ",
  "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.

```

GET /api HTTP/1.1
Host: backend.example.com
Authorization: Bearer eyJhbGciOiJFUzI1NiIsImtpZCI6IjllciJ9.eyJhdWQiOiJodHRwczovL2JhY2t1bmQuZXhhbXBsZS5jb20iLCJpc3MiOiJodHRwczovL2FzLmV4YW1wbGUuY29tIiwiaXNjaXNDQxOTE3NTkzLCJpYXQiOiJlbnN1YiI6ImJjQGV4YW1wbGUuY29tIiwic2NwIjpbImFwaSJdfQ.MXgnpvPMo0nhcePwnQbunD2gw_pDyCFA-Saobl6gyLAdyPbaALFuAOyFc4XTWaPEHV_LGmXklSTpz0yC7hlSQ

```

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) so the AS can parse it; for the latter it indicates what the AS issued it for (e.g. access\_token or refresh\_token).

This specification defines the token type identifiers "urn:ietf:params:oauth:token-type:access\_token" and "urn:ietf:params:oauth:token-type:refresh\_token" to indicate that the token is an OAuth 2.0 access token or refresh token, respectively. The value "urn:ietf:params:oauth:token-type:jwt" defined in Section 9 of [JWT] indicates that the token is a JWT. This specification also defines the token type identifier "urn:ietf:params:oauth:token-type:id\_token" to indicate that the token is an ID Token, as defined in Section 2 of [OpenID.Core]. Other URIs to indicate other token types MAY be used.

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]).

#### 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.

##### 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 must not be used.

The following example 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 following example 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 consumer.example.com-web-application is the current actor and admin@example.com was a prior actor. Such a token might come about as the result of the web application receiving a token like the one in the previous example and exchanging it for a new token that lists it as the current actor and that can be used at https://backend.example.com.

```
{
  "aud": "https://backend.example.com",
  "iss": "https://issuer.example.com",
  "exp": 1443904100,
  "nbf": 1443904000,
  "sub": "user@example.com",
  "act": {
    {
      "sub": "consumer.example.com-web-application",
      "iss": "https://issuer.example.net",
      "act": {
        {
          "sub": "admin@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 the claim of the same name.

#### 4.2. "scp" (Scopes) Claim

The "scp" claim is an array of strings, each of which represents an OAuth scope granted for the issued security token. Each array entry of the claim value is a scope-token, as defined in Section 3.3 of OAuth 2.0 [RFC6749].

The following example illustrates the "scp" claim within a JWT Claims Set with four scope-tokens.

```
{
  "aud": "https://consumer.example.com",
  "iss": "https://issuer.example.com",
  "exp": 1443904177,
  "nbf": 1443904077,
  "sub": "dgaf4mvfs75Fci_FL3heQA",
  "scp": ["email", "address", "profile", "phone"]
}
```

Figure 7: Scopes Claim

OAuth 2.0 Token Introspection [RFC7662] defines the "scope" parameter to convey the scopes associated with the token.

#### 4.3. "cid" (Client Identifier) Claim

The "cid" claim carries the client identifier of the OAuth 2.0 [RFC6749] client that requested the token.

The following example illustrates the "cid" 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",
  "cid": "s6BhdRkqt3"
}
```

Figure 8: Client Identifier Claim

OAuth 2.0 Token Introspection [RFC7662] defines the "client\_id" parameter as the client identifier for the OAuth 2.0 client that requested the token.

#### 4.4. "may\_act" (May Act For) 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 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 should not be used.

The following example 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: May Act For Claim

When included as a top-level member of an OAuth token introspection response, "may\_act" has the same semantics and format as the the claim of the same name.

## 5. IANA Considerations

### 5.1. OAuth URI Registration

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

#### 5.1.1. 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]]

## 5.2. OAuth Parameters Registration

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

### 5.2.1. 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 ]]

### 5.3. 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].

#### 5.3.1. 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 ]]

### 5.4. 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].

#### 5.4.1. 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: "scp"
- o Claim Description: Scope Values
- o Change Controller: IESG
- o Specification Document(s): Section 4.2 of [[ this specification ]]

- o Claim Name: "cid"
- 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: May Act For
- o Change Controller: IESG
- o Specification Document(s): Section 4.4 of [[ this specification ]]

#### 5.5. 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].

##### 5.5.1. 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: May Act For
- o Change Controller: IESG
- o Specification Document(s): Section 4.4 of [[ this specification ]]

#### 5.6. OAuth Extensions Error Registration

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

##### 5.6.1. 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 ]]

#### 6. Security Considerations

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 "scp" claim is suggested to mitigate potential for such abuse, as it restricts the contexts in which the delegated rights can be exercised.

## 7. Privacy Considerations

Tokens typically carry personal information and their usage in Token Exchange may reveal details of the target services being accessed. As such, tokens should only be requested and sent according to the privacy policies at the respective organizations.

## 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, <<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>>.
- [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>>.

## 8.2. Informative References

- [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", August 2015, <[http://openid.net/specs/openid-connect-core-1\\_0.html](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, <<http://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, <<http://www.rfc-editor.org/info/rfc6755>>.
- [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, <<http://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, <<http://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. 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=eyJhbGciOiJFUzI1NiIsImtpZCI6IjE2In0.eyJhdWQiOiJodHRwczovL2FzLmV4YW1wbGUuY29tIiwiaXNzIjoiaHR0cHM6Ly9vcmlnaW5hbC1pc3NlZXIuZXhhbXBsZS5uZXQiLCJleHAiOjE0NDE5MTA2MDAsIm5iZiI6MTQ0MTkwOTAwMCwic3ViIjoiYmNAZXhhbXBsZS5uZXQiLCJzY3AiOlsib3JkZXJzIiwicHJvZmlsZSI6Imhpc3RvcnkixX0.JDe7fZ267iIRXwbFmOugyCt5dmGoy6EeuzNQ3MqDek5cCUlyPhQC6cz9laKjKlbnjMQbLJqWix6ZdBI0isjsTA
&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 ("bc@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": "bc@example.net",
  "scp": ["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": "eyJhbGciOiJIJFZlNiIsImtpZCI6IjcyIn0.eyJhdWQiOiJlcm46ZXhhbXBsZTpjb29wZXh0IiwiaXNzIjoiaHR0cHM6Ly9hcy5leGFtcGxlLnV4cCI6MTQ0MTkxMzYxMCwic3ViIjoiyMNAZXXhhbXBsZS5uZXQiLCJzY3AiOlsib3JkZXJzIiwiaGlzdG9yeSI6InByb2ZpbGUuXX0.YQHULmI1YDTugbfEvgGY2gaGBmMyj9BepZSECCBE9j9ogqZv2qx6VQQPrbT1k7vBYGLNMOkkpmmmJkxZDS0YV7g",
  "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": "bc@example.net",
  "scp": ["orders", "history", "profile"]
}
```

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=eyJhbGciOiJFUzI1NiIsImtpZCI6IjE2In0.eyJhdWQiOiJodHRwczovL2FzLmV4YW1wbGUuY29tIiwiaXNzIjoiaHR0cHM6Ly9vcmlnaW5hbC1pc3NlZXIuZmZlZWVkbWluQGV4YW1wbGUubmV0Inl9.eyJhdWQiOiJodHRwczovL2FzLmV4YW1wbGUuY29tIiwiaXNzIjoiaHR0cHM6Ly9vcmlnaW5hbC1pc3NlZXIuZmZlZWVkbWluQGV4YW1wbGUubmV0Inl9.7YQ-3zPfhUvzje5oqw8COCvN5uP6NsKik9CVV6cAO4QKgM-tKfIOwgcZoUuDL2tEs6tqPlcBlmjiSzEjm3yBg
&actor_token=eyJhbGciOiJFUzI1NiIsImtpZCI6IjE2In0.eyJhdWQiOiJodHRwczovL2FzLmV4YW1wbGUuY29tIiwiaXNzIjoiaHR0cHM6Ly9vcmlnaW5hbC1pc3NlZXIuZmZlZWVkbWluQGV4YW1wbGUubmV0Inl9.7YQ-3zPfhUvzje5oqw8COCvN5uP6NsKik9CVV6cAO4QKgM-tKfIOwgcZoUuDL2tEs6tqPlcBlmjiSzEjm3yBg
&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,
  "scp": ["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": "eyJhbGciOiJFUzI1NiIsImtpZCI6IjcyIn0.eyJhdWQiOiJlcm46ZXhhbXBsZTpjb29wZXh0IiwiaXNzIjoiaHR0cHM6Ly9hcy5leGFtcGxlLmNvbSIsImV4cCI6MTQ0MTkxMzYxMCwic2NwIjpbInN0YXRlcylsImZlZlWQlXSXNlc3ViIjoidXNlckBleGFtcGxlLm5ldCIsImFjdCI6eyJzdWIiOiJhZG1pbkBlleGFtcGxlLm5ldCJ9fQ._qjM7Ij_HcrC78omT4jiZTFJOuzsAjlwPo3lymQS-Suqr64S1jCp6pfQR-in_OOAosAGamEg4jyPsht6kMAiYA",
  "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,
  "scp": ["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 and Derek Atkins with Kathleen Moriarty and Stephen Farrell serving as Security Area Directors. The following individuals contributed ideas, feedback, and wording to this specification:

Caleb Baker, Vittorio Bertocci, Thomas Broyer, William Denniss, Vladimir Dzhuvinov, Phil Hunt, Benjamin Kaduk, Jason Keglovitz, Torsten Lodderstedt, Adam Lewis, James Manger, Nov Mataka, Matt Miller, Matthew Perry, Justin Richer, Rifaat Shekh-Yusef, Scott Tomilson, and Hannes Tschofenig.

## Appendix C. Document History

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

-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](mailto:mbj@microsoft.com)  
URI: <http://self-issued.info/>

Anthony Nadalin  
Microsoft

Email: [tonynad@microsoft.com](mailto:tonynad@microsoft.com)

Brian Campbell (editor)  
Ping Identity

Email: [brian.d.campbell@gmail.com](mailto:brian.d.campbell@gmail.com)

John Bradley  
Ping Identity

Email: [ve7jtb@ve7jtb.com](mailto:ve7jtb@ve7jtb.com)

Chuck Mortimore  
Salesforce

Email: [cmortimore@salesforce.com](mailto:cmortimore@salesforce.com)

OAuth Working Group  
Internet-Draft  
Intended status: Best Current Practice  
Expires: January 4, 2018

Y. Sheffer  
Intuit  
D. Hardt  
Amazon  
M. Jones  
Microsoft  
July 03, 2017

JSON Web Token Best Current Practices  
draft-sheffer-oauth-jwt-bcp-01

Abstract

JSON Web Tokens, also known as JWTs [RFC7519], are URL-safe JSON-based security tokens that contain a set of claims that can be signed and/or encrypted. JWTs are being widely used and deployed as a simple security token format in numerous protocols and applications, both in the area of digital identity, and in other application areas. The goal of this Best Current Practices document is to provide actionable guidance leading to secure implementation and deployment of JWTs.

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 4, 2018.

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. Target Audience . . . . .	3
1.2. Conventions used in this document . . . . .	4
2. Threats and Vulnerabilities . . . . .	4
2.1. Weak Signatures and Insufficient Signature Validation . .	4
2.2. Weak symmetric keys . . . . .	4
2.3. Multiplicity of JSON encodings . . . . .	4
2.4. Incorrect Composition of Encryption and Signature . . . .	5
2.5. Insecure Use of Elliptic Curve Encryption . . . . .	5
2.6. Substitution Attacks . . . . .	5
2.7. Cross-JWT Confusion . . . . .	5
3. Best Practices . . . . .	5
3.1. Perform Algorithm Verification . . . . .	6
3.2. Use Appropriate Algorithms . . . . .	6
3.3. Validate All Cryptographic Operations . . . . .	6
3.4. Validate Cryptographic Inputs . . . . .	6
3.5. Ensure Cryptographic Keys have Sufficient Entropy . . . .	7
3.6. Use UTF-8 . . . . .	7
3.7. Validate Issuer and Subject . . . . .	7
3.8. Use and Validate Audience . . . . .	7
3.9. Use Explicit Typing . . . . .	8
3.10. Use Mutually Exclusive Validation Rules for Different Kinds of JWTs . . . . .	8
4. IANA Considerations . . . . .	9
5. Acknowledgements . . . . .	9
6. References . . . . .	9
6.1. Normative References . . . . .	9
6.2. Informative References . . . . .	10
Appendix A. Document History . . . . .	11
A.1. draft-sheffer-oauth-jwt-bcp-01 . . . . .	11
A.2. draft-sheffer-oauth-jwt-bcp-00 . . . . .	11
Authors' Addresses . . . . .	11

## 1. Introduction

JSON Web Tokens, also known as JWTs [RFC7519], are URL-safe JSON-based security tokens that contain a set of claims that can be signed and/or encrypted. The JWT specification has seen rapid adoption because it encapsulates security-relevant information in one, easy to

protect location, and because it is easy to implement using widely-available tools. One application area in which JWTs are commonly used is representing digital identity information, such as OpenID Connect ID Tokens [OpenID.Core] and OAuth 2.0 [RFC6749] access tokens and refresh tokens, the details of which are deployment-specific.

Since the JWT specification was published, there have been several widely published attacks on implementations and deployments. Such attacks are the result of under-specified security mechanisms, as well as incomplete implementations and incorrect usage by applications.

The goal of this document is to facilitate secure implementation and deployment of JWTs. Many of the recommendations in this document will actually be about implementation and use of the cryptographic mechanisms underlying JWTs that are defined by JSON Web Signature (JWS) [RFC7515], JSON Web Encryption (JWE) [RFC7516], and JSON Web Algorithms (JWA) [RFC7518]. Others will be about use of the JWT claims themselves.

These are intended to be minimum recommendations for the use of JWTs in the vast majority of implementation and deployment scenarios. Other specifications that reference this document can have stricter requirements related to one or more aspects of the format, based on their particular circumstances; when that is the case, implementers are advised to adhere to those stricter requirements. Furthermore, this document provides a floor, not a ceiling, so stronger options are always allowed (e.g., depending on differing evaluations of the importance of cryptographic strength vs. computational load).

Community knowledge about the strength of various algorithms and feasible attacks can change quickly, and experience shows that a Best Current Practice (BCP) document about security is a point-in-time statement. Readers are advised to seek out any errata or updates that apply to this document.

### 1.1. Target Audience

The targets of this document are:

- Implementers of JWT libraries (and the JWS and JWE libraries used by them),
- Implementers of code that uses such libraries (to the extent that some mechanisms may not be provided by libraries, or until they are), and

- Developers of specifications that rely on JWTs, both inside and outside the IETF.

## 1.2. Conventions used in this document

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 [RFC2119].

## 2. Threats and Vulnerabilities

This section lists some known and possible problems with JWT implementations and deployments. Each problem description is followed by references to one or more mitigations to those problems.

### 2.1. Weak Signatures and Insufficient Signature Validation

Signed JSON Web Tokens carry an explicit indication of the signing algorithm, in the form of the "alg" header parameter, to facilitate cryptographic agility. This, in conjunction with design flaws in some libraries and applications, have led to several attacks:

- The algorithm can be changed to "none" by an attacker, and some libraries would trust this value and "validate" the JWT without checking any signature.
- An "RS256" (RSA, 2048 bit) parameter value can be changed into "HS256" (HMAC, SHA-256), and some libraries would try to validate the signature using HMAC-SHA256 and using the RSA public key as the HMAC shared secret.

For mitigations, see Section 3.1 and Section 3.2.

### 2.2. Weak symmetric keys

In addition, some applications sign tokens using a weak symmetric key and a keyed MAC algorithm such as "HS256". In most cases, these keys are human memorable passwords that are vulnerable to dictionary attacks [Langkemper].

For mitigations, see Section 3.5.

### 2.3. Multiplicity of JSON encodings

Many practitioners are not aware that JSON [RFC7159] allows several different character encodings: UTF-8, UTF-16 and UTF-32. As a result, the JWT might be misinterpreted by its recipient.

For mitigations, see Section 3.6.

#### 2.4. Incorrect Composition of Encryption and Signature

Some libraries that decrypt a JWE-encrypted JWT to obtain a JWS-signed object do not always validate the internal signature.

For mitigations, see Section 3.3.

#### 2.5. Insecure Use of Elliptic Curve Encryption

Per [Sanso], several JOSE libraries fail to validate their inputs correctly when performing elliptic curve key agreement (the "ECDH-ES" algorithm). An attacker that is able to send JWEs of its choosing that use invalid curve points and observe the cleartext outputs resulting from decryption with the invalid curve points can use this vulnerability to recover the recipient's private key.

For mitigations, see Section 3.4.

#### 2.6. Substitution Attacks

There are attacks in which one recipient will have a JWT intended for it and attempt to use it at a different recipient that it was not intended for. If not caught, these attacks can result in the attacker gaining access to resources that it is not entitled to access.

For mitigations, see Section 3.7 and Section 3.8.

#### 2.7. Cross-JWT Confusion

As JWTs are being used by more different protocols in diverse application areas, it becomes increasingly important to prevent cases of JWT tokens that have been issued for one purpose being subverted and used for another. Note that this is a specific type of substitution attack. If the JWT could be used in an application context in which it could be confused with other kinds of JWTs, then mitigations MUST be employed to prevent these substitution attacks.

For mitigations, see Section 3.7, Section 3.8, Section 3.9, and Section 3.10.

### 3. Best Practices

The best practices listed below should be applied by practitioners to mitigate the threats listed in the preceding section.

### 3.1. Perform Algorithm Verification

Libraries MUST enable the caller to specify a supported set of algorithms and MUST NOT use any other algorithms when performing cryptographic operations. The library MUST ensure that the "alg" or "enc" header specifies the same algorithm that is used for the cryptographic operation. Moreover, each key MUST be used with exactly one algorithm, and this MUST be checked when the cryptographic operation is performed.

### 3.2. Use Appropriate Algorithms

As Section 5.2 of [RFC7515] says, "it is an application decision which algorithms may be used in a given context. Even if a JWS can be successfully validated, unless the algorithm(s) used in the JWS are acceptable to the application, it SHOULD consider the JWS to be invalid."

Therefore, applications MUST only allow the use of cryptographically current algorithms that meet the security requirements of the application. This set will vary over time as new algorithms are introduced and existing algorithms are deprecated due to discovered cryptographic weaknesses. Applications must therefore be designed to enable cryptographic agility.

That said, if a JWT is cryptographically protected by a transport layer, such as TLS using cryptographically current algorithms, there may be no need to apply another layer of cryptographic protections to the JWT. In such cases, the use of the "none" algorithm can be perfectly acceptable. JWTs using "none" are often used in application contexts in which the content is optionally signed; then the URL-safe claims representation and processing can be the same in both the signed and unsigned cases.

### 3.3. Validate All Cryptographic Operations

All cryptographic operations used in the JWT MUST be validated and the entire JWT MUST be rejected if any of them fail to validate. This is true not only of JWTs with a single set of Header Parameters but also for Nested JWTs, in which both outer and inner operations MUST be validated using the keys and algorithms supplied by the application.

### 3.4. Validate Cryptographic Inputs

Some cryptographic operations, such as Elliptic Curve Diffie-Hellman key agreement ("ECDH-ES") take inputs that may contain invalid values, such as points not on the specified elliptic curve or other

invalid points. Either the JWS/JWE library itself must validate these inputs before using them or it must use underlying cryptographic libraries that do so (or both!).

### 3.5. Ensure Cryptographic Keys have Sufficient Entropy

The Key Entropy and Random Values advice in Section 10.1 of [RFC7515] and the Password Considerations in Section 8.8 of [RFC7518] MUST be followed. In particular, human-memorizable passwords MUST NOT be directly used as the key to a keyed-MAC algorithm such as "HS256".

### 3.6. Use UTF-8

[RFC7515], [RFC7516], and [RFC7519] all specify that UTF-8 be used for encoding and decoding JSON used in Header Parameters and JWT Claims Sets. Implementations and applications MUST do this, and not use other Unicode encodings for these purposes.

### 3.7. Validate Issuer and Subject

When a JWT contains an "iss" (issuer) claim, the application MUST validate that the cryptographic keys used for the cryptographic operations in the JWT belong to the issuer. If they do not, the application MUST reject the JWT.

The means of determining the keys owned by an issuer is application-specific. As one example, OpenID Connect [OpenID.Core] issuer values are "https" URLs that reference a JSON metadata document that contains a "jwks\_uri" value that is an "https" URL from which the issuer's keys are retrieved as a JWK Set [RFC7517]. This same mechanism is used by [I-D.ietf-oauth-discovery]. Other applications may use different means of binding keys to issuers.

Similarly, when the JWT contains a "sub" (subject) claim, the application MUST validate that the subject value corresponds to a valid subject and/or issuer/subject pair at the application. This may include confirming that the issuer is trusted by the application. If the issuer, subject, or the pair are invalid, the application MUST reject the JWT.

### 3.8. Use and Validate Audience

If the same issuer can issue JWTs that are intended for use by more than one relying party or application, the JWT MUST contain an "aud" (audience) claim that can be used to determine whether the JWT is being used by an intended party or was substituted by an attacker at an unintended party. Furthermore, the relying party or application



MUST validate the audience value and if the audience value is not associated with the recipient, it MUST reject the JWT.

### 3.9. Use Explicit Typing

Confusion of one kind of JWT for another can be prevented by having all the kinds of JWTs that could otherwise potentially be confused include an explicit JWT type value and include checking the type value in their validation rules. Explicit JWT typing is accomplished by using the "typ" header parameter. For instance, the [I-D.ietf-secevent-token] specification uses the "application/secevent+jwt" media type to perform explicit typing of Security Event Tokens (SETs).

Per the definition of "typ" in Section 4.1.9 of [RFC7515], it is RECOMMENDED that the "application/" prefix be omitted from the "typ" value. Therefore, for example, the "typ" value used to explicitly include a type for a SET SHOULD be "secevent+jwt". When explicit typing is employed for a JWT, it is RECOMMENDED that a media type name of the format "application/example+jwt" be used, where "example" is replaced by the identifier for the specific kind of JWT.

Note that the use of explicit typing may not achieve disambiguation from existing kinds of JWTs, as the validation rules for existing kinds of JWTs often do not use the "typ" header parameter value. Explicit typing is RECOMMENDED for new uses of JWTs.

### 3.10. Use Mutually Exclusive Validation Rules for Different Kinds of JWTs

Each application of JWTs defines a profile specifying the required and optional JWT claims and the validation rules associated with them. If more than one kind of JWT can be issued by the same issuer, the validation rules for those JWTs MUST be written such that they are mutually exclusive, rejecting JWTs of the wrong kind. To prevent substitution of JWTs from one context into another, a number of strategies may be employed:

- Use explicit typing for different kinds of JWTs. Then the distinct "typ" values can be used to differentiate between the different kinds of JWTs.
- Use different sets of required claims or different required claim values. Then the validation rules for one kind of JWT will reject those with different claims or values.
- Use different sets of required header parameters or different required header parameter values. Then the validation rules for

one kind of JWT will reject those with different header parameters or values.

- Use different keys for different kinds of JWTs. Then the keys used to validate one kind of JWT will fail to validate other kinds of JWTs.
- Use different "aud" values for different uses of JWTs from the same issuer. Then audience validation will reject JWTs substituted into inappropriate contexts.
- Use different issuers for different kinds of JWTs. Then the distinct "iss" values can be used to segregate the different kinds of JWTs.

Given the broad diversity of JWT usage and applications, the best combination of types, required claims, values, header parameters, key usages, and issuers to differentiate among different kinds of JWTs will, in general, be application specific.

#### 4. IANA Considerations

This document requires no IANA actions.

#### 5. Acknowledgements

Thanks to Antonio Sanso for bringing the "ECDH-ES" invalid point attack to the attention of JWE and JWT implementers. Thanks to Nat Sakimura for advocating the use of explicit typing.

#### 6. References

##### 6.1. 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>>.
- [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>>.

- [RFC7516] Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)", RFC 7516, DOI 10.17487/RFC7516, May 2015, <<http://www.rfc-editor.org/info/rfc7516>>.
- [RFC7518] Jones, M., "JSON Web Algorithms (JWA)", RFC 7518, DOI 10.17487/RFC7518, May 2015, <<http://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, <<http://www.rfc-editor.org/info/rfc7519>>.

## 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-06 (work in progress), March 2017.
- [I-D.ietf-secevent-token]  
Hunt, P., Denniss, W., Ansari, M., and M. Jones, "Security Event Token (SET)", draft-ietf-secevent-token-02 (work in progress), June 2017.
- [Langkemper]  
Langkemper, S., "Attacking JWT Authentication", September 2016, <<https://www.sjoerdlangkemper.nl/2016/09/28/attacking-jwt-authentication/>>.
- [OpenID.Core]  
Sakimura, N., Bradley, J., Jones, M., Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0", November 2014, <[http://openid.net/specs/openid-connect-core-1\\_0.html](http://openid.net/specs/openid-connect-core-1_0.html)>.
- [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>>.
- [RFC7517] Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7517, May 2015, <<http://www.rfc-editor.org/info/rfc7517>>.
- [Sanso]  
Sanso, A., "Critical Vulnerability Uncovered in JSON Encryption", March 2017, <<https://blogs.adobe.com/security/2017/03/critical-vulnerability-uncovered-in-json-encryption.html>>.

## Appendix A. Document History

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

### A.1. draft-sheffer-oauth-jwt-bcp-01

- Added explicit typing.

### A.2. draft-sheffer-oauth-jwt-bcp-00

- Initial version.

## Authors' Addresses

Yaron Sheffer  
Intuit

EMail: [yaronf.ietf@gmail.com](mailto:yaronf.ietf@gmail.com)

Dick Hardt  
Amazon

EMail: [dick@amazon.com](mailto:dick@amazon.com)

Michael B. Jones  
Microsoft

EMail: [mbj@microsoft.com](mailto:mbj@microsoft.com)  
URI: <http://self-issued.info/>

OAuth Working Group  
Internet-Draft  
Intended status: Standards Track  
Expires: January 4, 2018

W. Denniss  
Google  
July 3, 2017

OAuth 2.0 Incremental Authorization  
draft-wdenniss-oauth-incremental-auth-00

Abstract

OAuth 2.0 authorization requests that include every scope the client might ever need can result in over-scoped authorization and a sub-optimal end-user consent experience. This specification enhances the OAuth 2.0 authorization protocol by adding incremental authorization, the ability to request specific authorization scopes as needed, when they're needed, removing the requirement to request every possible scope that might be needed upfront.

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 4, 2018.

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 . . . . .	2
3. Terminology . . . . .	2
4. Incremental Auth for Confidential Clients . . . . .	3
5. Incremental Auth for Public Clients . . . . .	3
6. IANA Considerations . . . . .	4
6.1. OAuth Parameters Registry . . . . .	4
7. Normative References . . . . .	5
Appendix A. Acknowledgements . . . . .	5
Appendix B. Document History . . . . .	5
Author's Address . . . . .	5

## 1. Introduction

OAuth 2.0 clients may offer multiple features that requiring user authorization, but commonly not every user will use each feature. Without incremental authentication, applications need to either request all the possible scopes they need upfront, potentially resulting in a bad user experience, or track each authorization grant separately, complicating development.

The goal of incremental authorization is to allow clients to request just the scopes they need, when they need them, while allowing them to store a single authorization grant for the user that contains the sum of the scopes granted. Thus, each new authorization request increments the scope of the authorization grant, without the client needing to track a separate authorization grant for each group of scopes.

## 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:

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

#### 4. Incremental Auth for Confidential Clients

For confidential clients, such as web servers that can keep secrets, the authorization endpoint SHOULD treat scopes that the user already granted differently on the consent user interface. Typically such scopes are hidden for new authorization requests, or at least there is an indication that the user already approved them.

By itself, this property of the authorization endpoint enables incremental authorization. The client can track every scope they've ever requested, and include those scopes on every new authorization request.

To avoid the need for confidential clients to re-request already authorized scopes, authorization servers MAY support an additional "include\_granted\_scopes" parameter in the authorization request. This parameter, enables the client to request tokens during the authorization grant exchange that represent the full scope of the user's grant to the application including any previous grants, without the app needing to track the scopes directly.

The client indicates they wish the new authorization grant to include previously granted scopes by sending the following additional parameter in the OAuth 2.0 Authorization Request (Section 4.1.1 of [RFC6749].) using the following additional parameter:

include\_granted\_scopes OPTIONAL. Either "true" or "false". When "true", the authorization server SHOULD include previously granted scopes for this client in the new authorization grant.

#### 5. Incremental Auth for Public Clients

Unlike with confidential clients, it is NOT RECOMMEND to automatically approve OAuth requests for public clients without user consent (see Section 10.2 of OAuth 2.0 [RFC6749]), thus authorization grants shouldn't contain previously authorized scopes in the manner described above for confidential clients.

Public clients (and confidential clients using this technique) should instead track the scopes for every authorization grant, and only request yet to be granted scopes during incremental authorization. In the past, this would result in multiple discrete authorization grants that would need to be tracked. To enable incrementing a single authorization grant for public clients, the client supplies their existing refresh token during the authorization code exchange,

and receives new authorization tokens with the scope of the previous and current authorization grants.

The client sends the previous refresh token in the OAuth 2.0 Access Token Request (Section 4.1.3 of [RFC6749].) using the following additional parameter:

`existing_grant` OPTIONAL. The refresh token from the existing authorization grant.

When processing the token exchange, in addition to the normal processing of such a request, the token endpoint MUST verify that token provided in the "existing\_grant" parameter is unexpired and unrevoked, and was issued to the same client id and relates to the same user as the current authorization grant. If this verification succeeds, the new refresh token issued in the Access Token Response (Section 4.1.4 of ) SHOULD include authorization for the scopes in the previous grant.

## 6. IANA Considerations

This specification makes a registration request as follows:

### 6.1. OAuth Parameters Registry

This specification registers the following parameters in the IANA OAuth Parameters registry defined in OAuth 2.0 [RFC6749].

- o Parameter name: `include_granted_scopes`
- o Parameter usage location: authorization request
- o Change controller: IESG
- o Specification document(s): this document
- o Parameter name: `existing_grant`
- o Parameter usage location: token request
- o Change controller: IESG
- o Specification document(s): this document



## 7. 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>>.
- [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>>.

## Appendix A. Acknowledgements

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

Yanna Wu, Marius Scurtescu, Jason Huang, Nicholas Watson, and Breno de Medeiros.

## Appendix B. Document History

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

-00

- o Initial draft based on the implementation of incremental and "appcremental" auth at Google.

## Author's Address

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

Email: [wdenniss@google.com](mailto:wdenniss@google.com)  
URI: <http://wdenniss.com/incremental-auth>