The Little Book of OAuth 2.0 RFCs

Aaron Parecki

The Little Book of OAuth 2.0 RFCs

Aaron Parecki

The Little Book of OAuth 2.0 RFCs

Compiled by Aaron Parecki

Copyright © 2022 Aaron Parecki

RFC Text Copyright © 2012-2022 IETF Trust

OAuth Logo by Chris Messina

While every precaution has been taken in the preparation of this book, the publisher and the author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 979-8-607-50395-6

22335.1516

Second Edition

Table of Contents

Introduction v

1. RFC 6749: The OAuth 2.0 Authorization Framework 7
2. RFC 6750: OAuth 2.0 Bearer Tokens 87
3. RFC 7636: Proof Key for Code Exchange (PKCE) 109
4. RFC 6819: OAuth 2.0 Threat Model and Security Considerations 133
5. RFC 8252: OAuth 2.0 for Native and Mobile Apps 207
6. Draft: OAuth 2.0 for Browser-Based Apps 231
7. Draft: OAuth 2.0 Security Best Current Practice 263
8. RFC 8628: OAuth 2.0 Device Authorization Grant 321
9. RFC 7009: OAuth 2.0 Token Revocation 345
10. RFC 7662: OAuth 2.0 Token Introspection 359
11. RFC 8414: OAuth 2.0 Authorization Server Metadata 379

Appendix: Advanced Extensions 405

Appendix: Related Communities 407

Appendix: Acknowledgments 409

Introduction

It's often a challenge to understand the entire OAuth landscape and how all the different RFCs fit together. OAuth is made up of many small building blocks, from the first RFC published in 2012 to many additional RFCs following it. The later additions to the spec often either fill in the underspecified parts of the first RFC, or cover additional use cases such as mobile phones or smart TVs that weren't originally addressed in the first RFC.

This book is a collection of the most critical RFCs you'll need to understand when building an OAuth client or server. They are ordered not sequentially, but instead in order of logical progression, each building on the last to provide new and better functionality as a whole.

Each chapter has a short introduction describing where this RFC fits within the stack and how it relates to the others that came before it. These RFCs are reproduced in their entirety, including parts that may have been obsoleted by later specifications.

Chapter 1

RFC 6749: The OAuth 2.0 Authorization Framework

RFC 6749 is the core OAuth 2.0 framework. This RFC describes various roles in OAuth, several different authorization flows, and provides some extension points to build upon. There are many aspects left unspecified that you'll need to decide when building a complete implementation. Many of these details have been documented as extension specs.

This spec defines ways applications can get an access token, but doesn't define how applications use access tokens or what format access tokens should be. The next RFC, RFC 6750, defines "Bearer Tokens" which have become the most common access token type in practice.

In the time since RFC 6749 has been published, much of the web and mobile landscape has changed, and some of the assumptions made no longer apply. The spec used to recommend the Implicit flow for both mobile and JavaScript apps, but since then, the PKCE extension improves security in both platforms and is now the recommended flow instead. The Password grant was included mostly as a compromise for migrating legacy systems to the OAuth framework, but is not a good option for new applications, and is prohibited in the latest version of the Security Best Current Practice.

Internet Engineering Task Force (IETF)   D. Hardt, Ed.
Request for Comments: 6749   Microsoft
Obsoletes: 5849   October 2012
Category: Standards Track
ISSN: 2070-1721

The OAuth 2.0 Authorization Framework

Abstract

The OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf. This specification replaces and obsoletes the OAuth 1.0 protocol described in RFC 5849.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc6749.

Copyright Notice

Copyright (c) 2012 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 4
   1.1. Roles 6
   1.2. Protocol Flow 7
   1.3. Authorization Grant 8
        1.3.1. Authorization Code 8
        1.3.2. Implicit 8
        1.3.3. Resource Owner Password Credentials 9
        1.3.4. Client Credentials 9
   1.4. Access Token 10
   1.5. Refresh Token 10
   1.6. TLS Version 12
   1.7. HTTP Redirections 12
   1.8. Interoperability 12
   1.9. Notational Conventions 13
2. Client Registration 13
   2.1. Client Types 14
   2.2. Client Identifier 15
   2.3. Client Authentication 16
        2.3.1. Client Password 16
        2.3.2. Other Authentication Methods 17
   2.4. Unregistered Clients 17
3. Protocol Endpoints 18
   3.1. Authorization Endpoint 18
        3.1.1. Response Type 19
        3.1.2. Redirection Endpoint 19
   3.2. Token Endpoint 21
        3.2.1. Client Authentication 22
   3.3. Access Token Scope 23
4. Obtaining Authorization 23
   4.1. Authorization Code Grant 24
        4.1.1. Authorization Request 25
        4.1.2. Authorization Response 26
        4.1.3. Access Token Request 29
        4.1.4. Access Token Response 30
   4.2. Implicit Grant 31
        4.2.1. Authorization Request 33
        4.2.2. Access Token Response 35
   4.3. Resource Owner Password Credentials Grant 37
        4.3.1. Authorization Request and Response 39
        4.3.2. Access Token Request 39
        4.3.3. Access Token Response 40
   4.4. Client Credentials Grant 40
        4.4.1. Authorization Request and Response 41
        4.4.2. Access Token Request 41
        4.4.3. Access Token Response 42
   4.5. Extension Grants 42

5. Issuing an Access Token 43
5.1. Successful Response 43
5.2. Error Response 45
6. Refreshing an Access Token 47
7. Accessing Protected Resources 48
7.1. Access Token Types 49
7.2. Error Response 49
8. Extensibility 50
8.1. Defining Access Token Types 50
8.2. Defining New Endpoint Parameters 50
8.3. Defining New Authorization Grant Types 51
8.4. Defining New Authorization Endpoint Response Types 51
8.5. Defining Additional Error Codes 51
9. Native Applications 52
10. Security Considerations 53
10.1. Client Authentication 53
10.2. Client Impersonation 54
10.3. Access Tokens 55
10.4. Refresh Tokens 55
10.5. Authorization Codes 56
10.6. Authorization Code Redirection URI Manipulation 56
10.7. Resource Owner Password Credentials 57
10.8. Request Confidentiality 58
10.9. Ensuring Endpoint Authenticity 58
10.10. Credentials-Guessing Attacks 58
10.11. Phishing Attacks 58
10.12. Cross-Site Request Forgery 59
10.13. Clickjacking 60
10.14. Code Injection and Input Validation 60
10.15. Open Redirectors 60
10.16. Misuse of Access Token to Impersonate Resource Owner in Implicit Flow 61
11. IANA Considerations 62
11.1. OAuth Access Token Types Registry 62
11.1.1. Registration Template 62
11.2. OAuth Parameters Registry 63
11.2.1. Registration Template 63
11.2.2. Initial Registry Contents 64
11.3. OAuth Authorization Endpoint Response Types Registry 66
11.3.1. Registration Template 66
11.3.2. Initial Registry Contents 67
11.4. OAuth Extensions Error Registry 67
11.4.1. Registration Template 68
12. References 68
12.1. Normative References 68
12.2. Informative References 70

Appendix A. Augmented Backus-Naur Form (ABNF) Syntax 71
A.1. "client_id" Syntax 71
A.2. "client_secret" Syntax 71
A.3. "response_type" Syntax 71
A.4. "scope" Syntax 72
A.5. "state" Syntax 72
A.6. "redirect_uri" Syntax 72
A.7. "error" Syntax 72
A.8. "error_description" Syntax 72
A.9. "error_uri" Syntax 72
A.10. "grant_type" Syntax 73
A.11. "code" Syntax 73
A.12. "access_token" Syntax 73
A.13. "token_type" Syntax 73
A.14. "expires_in" Syntax 73
A.15. "username" Syntax 73
A.16. "password" Syntax 73
A.17. "refresh_token" Syntax 74
A.18. Endpoint Parameter Syntax 74
Appendix B. Use of application/x-www-form-urlencoded Media Type ...74
Appendix C. Acknowledgements 75

1. Introduction

In the traditional client-server authentication model, the client requests an access-restricted resource (protected resource) on the server by authenticating with the server using the resource owner's credentials. In order to provide third-party applications access to restricted resources, the resource owner shares its credentials with the third party. This creates several problems and limitations:

- Third-party applications are required to store the resource owner's credentials for future use, typically a password in clear-text.

- Servers are required to support password authentication, despite the security weaknesses inherent in passwords.

- Third-party applications gain overly broad access to the resource owner's protected resources, leaving resource owners without any ability to restrict duration or access to a limited subset of resources.

- Resource owners cannot revoke access to an individual third party without revoking access to all third parties, and must do so by changing the third party's password.

o Compromise of any third-party application results in compromise of the end-user's password and all of the data protected by that password.

OAuth addresses these issues by introducing an authorization layer and separating the role of the client from that of the resource owner. In OAuth, the client requests access to resources controlled by the resource owner and hosted by the resource server, and is issued a different set of credentials than those of the resource owner.

Instead of using the resource owner's credentials to access protected resources, the client obtains an access token -- a string denoting a specific scope, lifetime, and other access attributes. Access tokens are issued to third-party clients by an authorization server with the approval of the resource owner. The client uses the access token to access the protected resources hosted by the resource server.

For example, an end-user (resource owner) can grant a printing service (client) access to her protected photos stored at a photo-sharing service (resource server), without sharing her username and password with the printing service. Instead, she authenticates directly with a server trusted by the photo-sharing service (authorization server), which issues the printing service delegation-specific credentials (access token).

This specification is designed for use with HTTP ([RFC2616]). The use of OAuth over any protocol other than HTTP is out of scope.

The OAuth 1.0 protocol ([RFC5849]), published as an informational document, was the result of a small ad hoc community effort. This Standards Track specification builds on the OAuth 1.0 deployment experience, as well as additional use cases and extensibility requirements gathered from the wider IETF community. The OAuth 2.0 protocol is not backward compatible with OAuth 1.0. The two versions may co-exist on the network, and implementations may choose to support both. However, it is the intention of this specification that new implementations support OAuth 2.0 as specified in this document and that OAuth 1.0 is used only to support existing deployments. The OAuth 2.0 protocol shares very few implementation details with the OAuth 1.0 protocol. Implementers familiar with OAuth 1.0 should approach this document without any assumptions as to its structure and details.

1.1. Roles

OAuth defines four roles:

resource owner
An entity capable of granting access to a protected resource. When the resource owner is a person, it is referred to as an end-user.

resource server
The server hosting the protected resources, capable of accepting and responding to protected resource requests using access tokens.

client
An application making protected resource requests on behalf of the resource owner and with its authorization. The term "client" does not imply any particular implementation characteristics (e.g., whether the application executes on a server, a desktop, or other devices).

authorization server
The server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization.

The interaction between the authorization server and resource server is beyond the scope of this specification. The authorization server may be the same server as the resource server or a separate entity. A single authorization server may issue access tokens accepted by multiple resource servers.

1.2. Protocol Flow

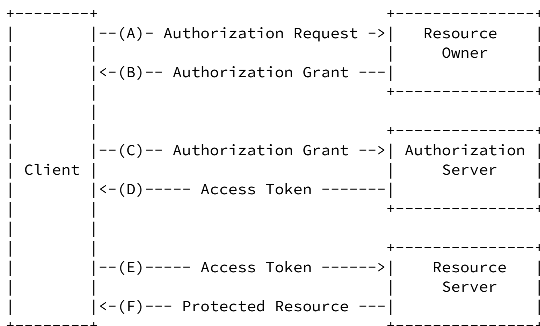


Figure 1: Abstract Protocol Flow

The abstract OAuth 2.0 flow illustrated in Figure 1 describes the interaction between the four roles and includes the following steps:

(A) The client requests authorization from the resource owner. The authorization request can be made directly to the resource owner (as shown), or preferably indirectly via the authorization server as an intermediary.

(B) The client receives an authorization grant, which is a credential representing the resource owner's authorization, expressed using one of four grant types defined in this specification or using an extension grant type. The authorization grant type depends on the method used by the client to request authorization and the types supported by the authorization server.

(C) The client requests an access token by authenticating with the authorization server and presenting the authorization grant.

(D) The authorization server authenticates the client and validates the authorization grant, and if valid, issues an access token.

(E) The client requests the protected resource from the resource server and authenticates by presenting the access token.

(F) The resource server validates the access token, and if valid, serves the request.

The preferred method for the client to obtain an authorization grant from the resource owner (depicted in steps (A) and (B)) is to use the authorization server as an intermediary, which is illustrated in Figure 3 in Section 4.1.

1.3. Authorization Grant

An authorization grant is a credential representing the resource owner's authorization (to access its protected resources) used by the client to obtain an access token. This specification defines four grant types -- authorization code, implicit, resource owner password credentials, and client credentials -- as well as an extensibility mechanism for defining additional types.

1.3.1. Authorization Code

The authorization code is obtained by using an authorization server as an intermediary between the client and resource owner. Instead of requesting authorization directly from the resource owner, the client directs the resource owner to an authorization server (via its user-agent as defined in [RFC2616]), which in turn directs the resource owner back to the client with the authorization code.

Before directing the resource owner back to the client with the authorization code, the authorization server authenticates the resource owner and obtains authorization. Because the resource owner only authenticates with the authorization server, the resource owner's credentials are never shared with the client.

The authorization code provides a few important security benefits, such as the ability to authenticate the client, as well as the transmission of the access token directly to the client without passing it through the resource owner's user-agent and potentially exposing it to others, including the resource owner.

1.3.2. Implicit

The implicit grant is a simplified authorization code flow optimized for clients implemented in a browser using a scripting language such as JavaScript. In the implicit flow, instead of issuing the client an authorization code, the client is issued an access token directly

(as the result of the resource owner authorization). The grant type is implicit, as no intermediate credentials (such as an authorization code) are issued (and later used to obtain an access token).

When issuing an access token during the implicit grant flow, the authorization server does not authenticate the client. In some cases, the client identity can be verified via the redirection URI used to deliver the access token to the client. The access token may be exposed to the resource owner or other applications with access to the resource owner's user-agent.

Implicit grants improve the responsiveness and efficiency of some clients (such as a client implemented as an in-browser application), since it reduces the number of round trips required to obtain an access token. However, this convenience should be weighed against the security implications of using implicit grants, such as those described in Sections 10.3 and 10.16, especially when the authorization code grant type is available.

1.3.3. Resource Owner Password Credentials

The resource owner password credentials (i.e., username and password) can be used directly as an authorization grant to obtain an access token. The credentials should only be used when there is a high degree of trust between the resource owner and the client (e.g., the client is part of the device operating system or a highly privileged application), and when other authorization grant types are not available (such as an authorization code).

Even though this grant type requires direct client access to the resource owner credentials, the resource owner credentials are used for a single request and are exchanged for an access token. This grant type can eliminate the need for the client to store the resource owner credentials for future use, by exchanging the credentials with a long-lived access token or refresh token.

1.3.4. Client Credentials

The client credentials (or other forms of client authentication) can be used as an authorization grant when the authorization scope is limited to the protected resources under the control of the client, or to protected resources previously arranged with the authorization server. Client credentials are used as an authorization grant typically when the client is acting on its own behalf (the client is also the resource owner) or is requesting access to protected resources based on an authorization previously arranged with the authorization server.

1.4. Access Token

Access tokens are credentials used to access protected resources. An access token is a string representing an authorization issued to the client. The string is usually opaque to the client. Tokens represent specific scopes and durations of access, granted by the resource owner, and enforced by the resource server and authorization server.

The token may denote an identifier used to retrieve the authorization information or may self-contain the authorization information in a verifiable manner (i.e., a token string consisting of some data and a signature). Additional authentication credentials, which are beyond the scope of this specification, may be required in order for the client to use a token.

The access token provides an abstraction layer, replacing different authorization constructs (e.g., username and password) with a single token understood by the resource server. This abstraction enables issuing access tokens more restrictive than the authorization grant used to obtain them, as well as removing the resource server's need to understand a wide range of authentication methods.

Access tokens can have different formats, structures, and methods of utilization (e.g., cryptographic properties) based on the resource server security requirements. Access token attributes and the methods used to access protected resources are beyond the scope of this specification and are defined by companion specifications such as [RFC6750].

1.5. Refresh Token

Refresh tokens are credentials used to obtain access tokens. Refresh tokens are issued to the client by the authorization server and are used to obtain a new access token when the current access token becomes invalid or expires, or to obtain additional access tokens with identical or narrower scope (access tokens may have a shorter lifetime and fewer permissions than authorized by the resource owner). Issuing a refresh token is optional at the discretion of the authorization server. If the authorization server issues a refresh token, it is included when issuing an access token (i.e., step (D) in Figure 1).

A refresh token is a string representing the authorization granted to the client by the resource owner. The string is usually opaque to the client. The token denotes an identifier used to retrieve the

authorization information. Unlike access tokens, refresh tokens are intended for use only with authorization servers and are never sent to resource servers.

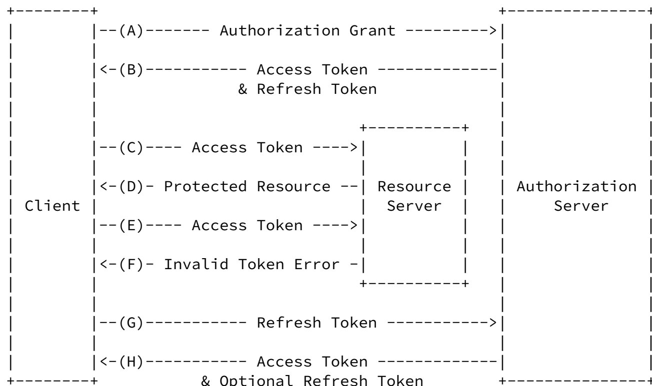


Figure 2: Refreshing an Expired Access Token

The flow illustrated in Figure 2 includes the following steps:

(A) The client requests an access token by authenticating with the authorization server and presenting an authorization grant.

(B) The authorization server authenticates the client and validates the authorization grant, and if valid, issues an access token and a refresh token.

(C) The client makes a protected resource request to the resource server by presenting the access token.

(D) The resource server validates the access token, and if valid, serves the request.

(E) Steps (C) and (D) repeat until the access token expires. If the client knows the access token expired, it skips to step (G); otherwise, it makes another protected resource request.

(F) Since the access token is invalid, the resource server returns an invalid token error.

(G) The client requests a new access token by authenticating with the authorization server and presenting the refresh token. The client authentication requirements are based on the client type and on the authorization server policies.

(H) The authorization server authenticates the client and validates the refresh token, and if valid, issues a new access token (and, optionally, a new refresh token).

Steps (C), (D), (E), and (F) are outside the scope of this specification, as described in Section 7.

1.6. TLS Version

Whenever Transport Layer Security (TLS) is used by this specification, the appropriate version (or versions) of TLS will vary over time, based on the widespread deployment and known security vulnerabilities. At the time of this writing, TLS version 1.2 [RFC5246] is the most recent version, but has a very limited deployment base and might not be readily available for implementation. TLS version 1.0 [RFC2246] is the most widely deployed version and will provide the broadest interoperability.

Implementations MAY also support additional transport-layer security mechanisms that meet their security requirements.

1.7. HTTP Redirections

This specification makes extensive use of HTTP redirections, in which the client or the authorization server directs the resource owner's user-agent to another destination. While the examples in this specification show the use of the HTTP 302 status code, any other method available via the user-agent to accomplish this redirection is allowed and is considered to be an implementation detail.

1.8. Interoperability

OAuth 2.0 provides a rich authorization framework with well-defined security properties. However, as a rich and highly extensible framework with many optional components, on its own, this specification is likely to produce a wide range of non-interoperable implementations.

In addition, this specification leaves a few required components partially or fully undefined (e.g., client registration, authorization server capabilities, endpoint discovery). Without

these components, clients must be manually and specifically configured against a specific authorization server and resource server in order to interoperate.

This framework was designed with the clear expectation that future work will define prescriptive profiles and extensions necessary to achieve full web-scale interoperability.

1.9. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this specification are to be interpreted as described in [RFC2119].

This specification uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234]. Additionally, the rule URI-reference is included from "Uniform Resource Identifier (URI): Generic Syntax" [RFC3986].

Certain security-related terms are to be understood in the sense defined in [RFC4949]. These terms include, but are not limited to, "attack", "authentication", "authorization", "certificate", "confidentiality", "credential", "encryption", "identity", "sign", "signature", "trust", "validate", and "verify".

Unless otherwise noted, all the protocol parameter names and values are case sensitive.

2. Client Registration

Before initiating the protocol, the client registers with the authorization server. The means through which the client registers with the authorization server are beyond the scope of this specification but typically involve end-user interaction with an HTML registration form.

Client registration does not require a direct interaction between the client and the authorization server. When supported by the authorization server, registration can rely on other means for establishing trust and obtaining the required client properties (e.g., redirection URI, client type). For example, registration can be accomplished using a self-issued or third-party-issued assertion, or by the authorization server performing client discovery using a trusted channel.

When registering a client, the client developer SHALL:

- o specify the client type as described in Section 2.1,
- o provide its client redirection URIs as described in Section 3.1.2, and
- o include any other information required by the authorization server (e.g., application name, website, description, logo image, the acceptance of legal terms).

2.1. Client Types

OAuth defines two client types, based on their ability to authenticate securely with the authorization server (i.e., ability to maintain the confidentiality of their client credentials):

confidential
: Clients capable of maintaining the confidentiality of their credentials (e.g., client implemented on a secure server with restricted access to the client credentials), or capable of secure client authentication using other means.

public
: Clients incapable of maintaining the confidentiality of their credentials (e.g., clients executing on the device used by the resource owner, such as an installed native application or a web browser-based application), and incapable of secure client authentication via any other means.

The client type designation is based on the authorization server's definition of secure authentication and its acceptable exposure levels of client credentials. The authorization server SHOULD NOT make assumptions about the client type.

A client may be implemented as a distributed set of components, each with a different client type and security context (e.g., a distributed client with both a confidential server-based component and a public browser-based component). If the authorization server does not provide support for such clients or does not provide guidance with regard to their registration, the client SHOULD register each component as a separate client.

This specification has been designed around the following client profiles:

web application
A web application is a confidential client running on a web server. Resource owners access the client via an HTML user interface rendered in a user-agent on the device used by the resource owner. The client credentials as well as any access token issued to the client are stored on the web server and are not exposed to or accessible by the resource owner.

user-agent-based application
A user-agent-based application is a public client in which the client code is downloaded from a web server and executes within a user-agent (e.g., web browser) on the device used by the resource owner. Protocol data and credentials are easily accessible (and often visible) to the resource owner. Since such applications reside within the user-agent, they can make seamless use of the user-agent capabilities when requesting authorization.

native application
A native application is a public client installed and executed on the device used by the resource owner. Protocol data and credentials are accessible to the resource owner. It is assumed that any client authentication credentials included in the application can be extracted. On the other hand, dynamically issued credentials such as access tokens or refresh tokens can receive an acceptable level of protection. At a minimum, these credentials are protected from hostile servers with which the application may interact. On some platforms, these credentials might be protected from other applications residing on the same device.

2.2. Client Identifier

The authorization server issues the registered client a client identifier -- a unique string representing the registration information provided by the client. The client identifier is not a secret; it is exposed to the resource owner and MUST NOT be used alone for client authentication. The client identifier is unique to the authorization server.

The client identifier string size is left undefined by this specification. The client should avoid making assumptions about the identifier size. The authorization server SHOULD document the size of any identifier it issues.

2.3. Client Authentication

If the client type is confidential, the client and authorization server establish a client authentication method suitable for the security requirements of the authorization server. The authorization server MAY accept any form of client authentication meeting its security requirements.

Confidential clients are typically issued (or establish) a set of client credentials used for authenticating with the authorization server (e.g., password, public/private key pair).

The authorization server MAY establish a client authentication method with public clients. However, the authorization server MUST NOT rely on public client authentication for the purpose of identifying the client.

The client MUST NOT use more than one authentication method in each request.

2.3.1. Client Password

Clients in possession of a client password MAY use the HTTP Basic authentication scheme as defined in [RFC2617] to authenticate with the authorization server. The client identifier is encoded using the "application/x-www-form-urlencoded" encoding algorithm per Appendix B, and the encoded value is used as the username; the client password is encoded using the same algorithm and used as the password. The authorization server MUST support the HTTP Basic authentication scheme for authenticating clients that were issued a client password.

For example (with extra line breaks for display purposes only):

```
Authorization: Basic czZCaGRSa3F0Mzo3RmpmcDBaQnIxS3REUmJuZlZkbUl3
```

Alternatively, the authorization server MAY support including the client credentials in the request-body using the following parameters:

client_id
: REQUIRED. The client identifier issued to the client during the registration process described by Section 2.2.

client_secret
: REQUIRED. The client secret. The client MAY omit the parameter if the client secret is an empty string.

Including the client credentials in the request-body using the two parameters is NOT RECOMMENDED and SHOULD be limited to clients unable to directly utilize the HTTP Basic authentication scheme (or other password-based HTTP authentication schemes). The parameters can only be transmitted in the request-body and MUST NOT be included in the request URI.

For example, a request to refresh an access token (Section 6) using the body parameters (with extra line breaks for display purposes only):

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

grant_type=refresh_token&refresh_token=tGzv3JOkF0XG5Qx2TlKWIA
&client_id=s6BhdRkqt3&client_secret=7Fjfp0ZBr1KtDRbnfVdmIw
```

The authorization server MUST require the use of TLS as described in Section 1.6 when sending requests using password authentication.

Since this client authentication method involves a password, the authorization server MUST protect any endpoint utilizing it against brute force attacks.

2.3.2. Other Authentication Methods

The authorization server MAY support any suitable HTTP authentication scheme matching its security requirements. When using other authentication methods, the authorization server MUST define a mapping between the client identifier (registration record) and authentication scheme.

2.4. Unregistered Clients

This specification does not exclude the use of unregistered clients. However, the use of such clients is beyond the scope of this specification and requires additional security analysis and review of its interoperability impact.

3. Protocol Endpoints

The authorization process utilizes two authorization server endpoints (HTTP resources):

- Authorization endpoint - used by the client to obtain authorization from the resource owner via user-agent redirection.

- Token endpoint - used by the client to exchange an authorization grant for an access token, typically with client authentication.

As well as one client endpoint:

- Redirection endpoint - used by the authorization server to return responses containing authorization credentials to the client via the resource owner user-agent.

Not every authorization grant type utilizes both endpoints. Extension grant types MAY define additional endpoints as needed.

3.1. Authorization Endpoint

The authorization endpoint is used to interact with the resource owner and obtain an authorization grant. The authorization server MUST first verify the identity of the resource owner. The way in which the authorization server authenticates the resource owner (e.g., username and password login, session cookies) is beyond the scope of this specification.

The means through which the client obtains the location of the authorization endpoint are beyond the scope of this specification, but the location is typically provided in the service documentation.

The endpoint URI MAY include an "application/x-www-form-urlencoded" formatted (per Appendix B) query component ([RFC3986] Section 3.4), which MUST be retained when adding additional query parameters. The endpoint URI MUST NOT include a fragment component.

Since requests to the authorization endpoint result in user authentication and the transmission of clear-text credentials (in the HTTP response), the authorization server MUST require the use of TLS as described in Section 1.6 when sending requests to the authorization endpoint.

The authorization server MUST support the use of the HTTP "GET" method [RFC2616] for the authorization endpoint and MAY support the use of the "POST" method as well.

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.

3.1.1. Response Type

The authorization endpoint is used by the authorization code grant type and implicit grant type flows. The client informs the authorization server of the desired grant type using the following parameter:

response_type
: REQUIRED. The value MUST be one of "code" for requesting an authorization code as described by Section 4.1.1, "token" for requesting an access token (implicit grant) as described by Section 4.2.1, or a registered extension value as described by Section 8.4.

Extension response types MAY contain a space-delimited (%x20) list of values, where the order of values does not matter (e.g., response type "a b" is the same as "b a"). The meaning of such composite response types is defined by their respective specifications.

If an authorization request is missing the "response_type" parameter, or if the response type is not understood, the authorization server MUST return an error response as described in Section 4.1.2.1.

3.1.2. Redirection Endpoint

After completing its interaction with the resource owner, the authorization server directs the resource owner's user-agent back to the client. The authorization server redirects the user-agent to the client's redirection endpoint previously established with the authorization server during the client registration process or when making the authorization request.

The redirection endpoint URI MUST be an absolute URI as defined by [RFC3986] Section 4.3. The endpoint URI MAY include an "application/x-www-form-urlencoded" formatted (per Appendix B) query component ([RFC3986] Section 3.4), which MUST be retained when adding additional query parameters. The endpoint URI MUST NOT include a fragment component.

3.1.2.1. Endpoint Request Confidentiality

The redirection endpoint SHOULD require the use of TLS as described in Section 1.6 when the requested response type is "code" or "token", or when the redirection request will result in the transmission of sensitive credentials over an open network. This specification does not mandate the use of TLS because at the time of this writing, requiring clients to deploy TLS is a significant hurdle for many client developers. If TLS is not available, the authorization server SHOULD warn the resource owner about the insecure endpoint prior to redirection (e.g., display a message during the authorization request).

Lack of transport-layer security can have a severe impact on the security of the client and the protected resources it is authorized to access. The use of transport-layer security is particularly critical when the authorization process is used as a form of delegated end-user authentication by the client (e.g., third-party sign-in service).

3.1.2.2. Registration Requirements

The authorization server MUST require the following clients to register their redirection endpoint:

- o Public clients.

- o Confidential clients utilizing the implicit grant type.

The authorization server SHOULD require all clients to register their redirection endpoint prior to utilizing the authorization endpoint.

The authorization server SHOULD require the client to provide the complete redirection URI (the client MAY use the "state" request parameter to achieve per-request customization). If requiring the registration of the complete redirection URI is not possible, the authorization server SHOULD require the registration of the URI scheme, authority, and path (allowing the client to dynamically vary only the query component of the redirection URI when requesting authorization).

The authorization server MAY allow the client to register multiple redirection endpoints.

Lack of a redirection URI registration requirement can enable an attacker to use the authorization endpoint as an open redirector as described in Section 10.15.

3.1.2.3. Dynamic Configuration

If multiple redirection URIs have been registered, if only part of the redirection URI has been registered, or if no redirection URI has been registered, the client MUST include a redirection URI with the authorization request using the "redirect_uri" request parameter.

When a redirection URI is included in an authorization request, the authorization server MUST compare and match the value received against at least one of the registered redirection URIs (or URI components) as defined in [RFC3986] Section 6, if any redirection URIs were registered. If the client registration included the full redirection URI, the authorization server MUST compare the two URIs using simple string comparison as defined in [RFC3986] Section 6.2.1.

3.1.2.4. Invalid Endpoint

If an authorization request fails validation due to a missing, invalid, or mismatching redirection URI, the authorization server SHOULD inform the resource owner of the error and MUST NOT automatically redirect the user-agent to the invalid redirection URI.

3.1.2.5. Endpoint Content

The redirection request to the client's endpoint typically results in an HTML document response, processed by the user-agent. If the HTML response is served directly as the result of the redirection request, any script included in the HTML document will execute with full access to the redirection URI and the credentials it contains.

The client SHOULD NOT include any third-party scripts (e.g., third-party analytics, social plug-ins, ad networks) in the redirection endpoint response. Instead, it SHOULD extract the credentials from the URI and redirect the user-agent again to another endpoint without exposing the credentials (in the URI or elsewhere). If third-party scripts are included, the client MUST ensure that its own scripts (used to extract and remove the credentials from the URI) will execute first.

3.2. Token Endpoint

The token endpoint is used by the client to obtain an access token by presenting its authorization grant or refresh token. The token endpoint is used with every authorization grant except for the implicit grant type (since an access token is issued directly).

The means through which the client obtains the location of the token endpoint are beyond the scope of this specification, but the location is typically provided in the service documentation.

The endpoint URI MAY include an "application/x-www-form-urlencoded" formatted (per Appendix B) query component ([RFC3986] Section 3.4), which MUST be retained when adding additional query parameters. The endpoint URI MUST NOT include a fragment component.

Since requests to the token endpoint result in the transmission of clear-text credentials (in the HTTP request and response), the authorization server MUST require the use of TLS as described in Section 1.6 when sending requests to the token endpoint.

The client MUST use the HTTP "POST" method when making access token requests.

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.

3.2.1. Client Authentication

Confidential clients or other clients issued client credentials MUST authenticate with the authorization server as described in Section 2.3 when making requests to the token endpoint. Client authentication is used for:

- Enforcing the binding of refresh tokens and authorization codes to the client they were issued to. Client authentication is critical when an authorization code is transmitted to the redirection endpoint over an insecure channel or when the redirection URI has not been registered in full.

- Recovering from a compromised client by disabling the client or changing its credentials, thus preventing an attacker from abusing stolen refresh tokens. Changing a single set of client credentials is significantly faster than revoking an entire set of refresh tokens.

- Implementing authentication management best practices, which require periodic credential rotation. Rotation of an entire set of refresh tokens can be challenging, while rotation of a single set of client credentials is significantly easier.

A client MAY use the "client_id" request parameter to identify itself when sending requests to the token endpoint. In the "authorization_code" "grant_type" request to the token endpoint, an unauthenticated client MUST send its "client_id" to prevent itself from inadvertently accepting a code intended for a client with a different "client_id". This protects the client from substitution of the authentication code. (It provides no additional security for the protected resource.)

3.3. Access Token Scope

The authorization and token endpoints allow the client to specify the scope of the access request using the "scope" request parameter. In turn, the authorization server uses the "scope" response parameter to inform the client of the scope of the access token issued.

The value of the scope parameter is expressed as a list of space-delimited, case-sensitive strings. The strings are defined by the authorization server. If the value contains multiple space-delimited strings, their order does not matter, and each string adds an additional access range to the requested scope.

```
scope       = scope-token *( SP scope-token )
scope-token = 1*( %x21 / %x23-5B / %x5D-7E )
```

The authorization server MAY fully or partially ignore the scope requested by the client, based on the authorization server policy or the resource owner's instructions. If the issued access token scope is different from the one requested by the client, the authorization server MUST include the "scope" response parameter to inform the client of the actual scope granted.

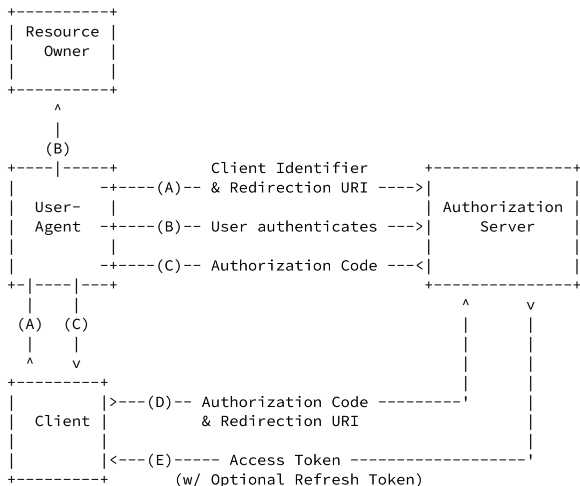
If the client omits the scope parameter when requesting authorization, the authorization server MUST either process the request using a pre-defined default value or fail the request indicating an invalid scope. The authorization server SHOULD document its scope requirements and default value (if defined).

4. Obtaining Authorization

To request an access token, the client obtains authorization from the resource owner. The authorization is expressed in the form of an authorization grant, which the client uses to request the access token. OAuth defines four grant types: authorization code, implicit, resource owner password credentials, and client credentials. It also provides an extension mechanism for defining additional grant types.

4.1. Authorization Code Grant

The authorization code grant type is used to obtain both access tokens and refresh tokens and is optimized for confidential clients. Since this is a redirection-based flow, the client must be capable of interacting with the resource owner's user-agent (typically a web browser) and capable of receiving incoming requests (via redirection) from the authorization server.



Note: The lines illustrating steps (A), (B), and (C) are broken into two parts as they pass through the user-agent.

Figure 3: Authorization Code Flow

The flow illustrated in Figure 3 includes the following steps:

(A) The client initiates the flow by directing the resource owner's user-agent to the authorization endpoint. The client includes its client identifier, requested scope, local state, and a redirection URI to which the authorization server will send the user-agent back once access is granted (or denied).

(B) The authorization server authenticates the resource owner (via the user-agent) and establishes whether the resource owner grants or denies the client's access request.

(C) Assuming the resource owner grants access, the authorization server redirects the user-agent back to the client using the redirection URI provided earlier (in the request or during client registration). The redirection URI includes an authorization code and any local state provided by the client earlier.

(D) The client requests an access token from the authorization server's token endpoint by including the authorization code received in the previous step. When making the request, the client authenticates with the authorization server. The client includes the redirection URI used to obtain the authorization code for verification.

(E) The authorization server authenticates the client, validates the authorization code, and ensures that the redirection URI received matches the URI used to redirect the client in step (C). If valid, the authorization server responds back with an access token and, optionally, a refresh token.

4.1.1. Authorization Request

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

response_type
: REQUIRED. Value MUST be set to "code".

client_id
: REQUIRED. The client identifier as described in Section 2.2.

redirect_uri
: OPTIONAL. As described in Section 3.1.2.

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

state
RECOMMENDED. An opaque value used by the client to maintain state between the request and callback. The authorization server includes this value when redirecting the user-agent back to the client. The parameter SHOULD be used for preventing cross-site request forgery as described in Section 10.12.

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

For example, the client directs the user-agent to make the following HTTP request using TLS (with extra line breaks for display purposes only):

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

The authorization server validates the request to ensure that all required parameters are present and valid. If the request is valid, the authorization server authenticates the resource owner and obtains an authorization decision (by asking the resource owner or by establishing approval via other means).

When a decision is established, the authorization server directs the user-agent to the provided client redirection URI using an HTTP redirection response, or by other means available to it via the user-agent.

4.1.2. Authorization Response

If the resource owner grants the access request, the authorization server issues an authorization code and delivers it to the client by adding the following parameters to the query component of the redirection URI using the "application/x-www-form-urlencoded" format, per Appendix B:

code
REQUIRED. The authorization code generated by the authorization server. The authorization code MUST expire shortly after it is issued to mitigate the risk of leaks. A maximum authorization code lifetime of 10 minutes is RECOMMENDED. The client MUST NOT use the authorization code

more than once. If an authorization code is used more than once, the authorization server MUST deny the request and SHOULD revoke (when possible) all tokens previously issued based on that authorization code. The authorization code is bound to the client identifier and redirection URI.

state
REQUIRED if the "state" parameter was present in the client authorization request. The exact value received from the client.

For example, the authorization server redirects the user-agent by sending the following HTTP response:

```
HTTP/1.1 302 Found
Location: https://client.example.com/cb?code=SplxlOBeZQQYbYS6WxSbIA
          &state=xyz
```

The client MUST ignore unrecognized response parameters. The authorization code string size is left undefined by this specification. The client should avoid making assumptions about code value sizes. The authorization server SHOULD document the size of any value it issues.

4.1.2.1. Error Response

If the request fails due to a missing, invalid, or mismatching redirection URI, or if the client identifier is missing or invalid, the authorization server SHOULD inform the resource owner of the error and MUST NOT automatically redirect the user-agent to the invalid redirection URI.

If the resource owner denies the access request or if the request fails for reasons other than a missing or invalid redirection URI, the authorization server informs the client by adding the following parameters to the query component of the redirection URI using the "application/x-www-form-urlencoded" format, per Appendix B:

error
REQUIRED. A single ASCII [USASCII] error code from the following:

invalid_request
The request is missing a required parameter, includes an invalid parameter value, includes a parameter more than once, or is otherwise malformed.

unauthorized_client
The client is not authorized to request an authorization code using this method.

access_denied
The resource owner or authorization server denied the request.

unsupported_response_type
The authorization server does not support obtaining an authorization code using this method.

invalid_scope
The requested scope is invalid, unknown, or malformed.

server_error
The authorization server encountered an unexpected condition that prevented it from fulfilling the request. (This error code is needed because a 500 Internal Server Error HTTP status code cannot be returned to the client via an HTTP redirect.)

temporarily_unavailable
The authorization server is currently unable to handle the request due to a temporary overloading or maintenance of the server. (This error code is needed because a 503 Service Unavailable HTTP status code cannot be returned to the client via an HTTP redirect.)

Values for the "error" parameter MUST NOT include characters outside the set %x20-21 / %x23-5B / %x5D-7E.

error_description
OPTIONAL. Human-readable ASCII [USASCII] text providing additional information, used to assist the client developer in understanding the error that occurred.
Values for the "error_description" parameter MUST NOT include characters outside the set %x20-21 / %x23-5B / %x5D-7E.

error_uri
OPTIONAL. A URI identifying a human-readable web page with information about the error, used to provide the client developer with additional information about the error.
Values for the "error_uri" parameter MUST conform to the URI-reference syntax and thus MUST NOT include characters outside the set %x21 / %x23-5B / %x5D-7E.

state
REQUIRED if a "state" parameter was present in the client authorization request. The exact value received from the client.

For example, the authorization server redirects the user-agent by sending the following HTTP response:

```
HTTP/1.1 302 Found
Location: https://client.example.com/cb?error=access_denied&state=xyz
```

4.1.3. Access Token Request

The client makes a request to the token endpoint by sending the following parameters using the "application/x-www-form-urlencoded" format per Appendix B with a character encoding of UTF-8 in the HTTP request entity-body:

grant_type
REQUIRED. Value MUST be set to "authorization_code".

code
REQUIRED. The authorization code received from the authorization server.

redirect_uri
REQUIRED, if the "redirect_uri" parameter was included in the authorization request as described in Section 4.1.1, and their values MUST be identical.

client_id
REQUIRED, if the client is not authenticating with the authorization server as described in Section 3.2.1.

If the client type is confidential or the client was issued client credentials (or assigned other authentication requirements), the client MUST authenticate with the authorization server as described in Section 3.2.1.

For example, the client makes the following HTTP request using TLS (with extra line breaks for display purposes only):

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

grant_type=authorization_code&code=SplxlOBeZQQYbYS6WxSbIA
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
```

The authorization server MUST:

- o require client authentication for confidential clients or for any client that was issued client credentials (or with other authentication requirements),
- o authenticate the client if client authentication is included,
- o ensure that the authorization code was issued to the authenticated confidential client, or if the client is public, ensure that the code was issued to "client_id" in the request,
- o verify that the authorization code is valid, and
- o 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.

4.1.4. Access Token Response

If the access token request is valid and authorized, the authorization server issues an access token and optional refresh token as described in Section 5.1. If the request client authentication failed or is invalid, the authorization server returns an error response as described in Section 5.2.

An example successful response:

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "access_token":"2YotnFZFEjr1zCsicMWpAA",
  "token_type":"example",
  "expires_in":3600,
  "refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
  "example_parameter":"example_value"
}
```

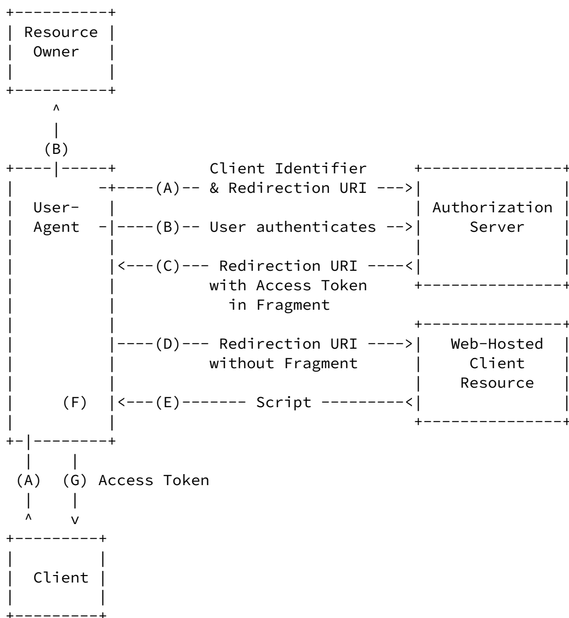
4.2. Implicit Grant

The implicit grant type is used to obtain access tokens (it does not support the issuance of refresh tokens) and is optimized for public clients known to operate a particular redirection URI. These clients are typically implemented in a browser using a scripting language such as JavaScript.

Since this is a redirection-based flow, the client must be capable of interacting with the resource owner's user-agent (typically a web browser) and capable of receiving incoming requests (via redirection) from the authorization server.

Unlike the authorization code grant type, in which the client makes separate requests for authorization and for an access token, the client receives the access token as the result of the authorization request.

The implicit grant type does not include client authentication, and relies on the presence of the resource owner and the registration of the redirection URI. Because the access token is encoded into the redirection URI, it may be exposed to the resource owner and other applications residing on the same device.



Note: The lines illustrating steps (A) and (B) are broken into two parts as they pass through the user-agent.

Figure 4: Implicit Grant Flow

The flow illustrated in Figure 4 includes the following steps:

(A) The client initiates the flow by directing the resource owner's user-agent to the authorization endpoint. The client includes its client identifier, requested scope, local state, and a redirection URI to which the authorization server will send the user-agent back once access is granted (or denied).

(B) The authorization server authenticates the resource owner (via the user-agent) and establishes whether the resource owner grants or denies the client's access request.

(C) Assuming the resource owner grants access, the authorization server redirects the user-agent back to the client using the redirection URI provided earlier. The redirection URI includes the access token in the URI fragment.

(D) The user-agent follows the redirection instructions by making a request to the web-hosted client resource (which does not include the fragment per [RFC2616]). The user-agent retains the fragment information locally.

(E) The web-hosted client resource returns a web page (typically an HTML document with an embedded script) capable of accessing the full redirection URI including the fragment retained by the user-agent, and extracting the access token (and other parameters) contained in the fragment.

(F) The user-agent executes the script provided by the web-hosted client resource locally, which extracts the access token.

(G) The user-agent passes the access token to the client.

See Sections 1.3.2 and 9 for background on using the implicit grant. See Sections 10.3 and 10.16 for important security considerations when using the implicit grant.

4.2.1. Authorization Request

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

response_type
REQUIRED. Value MUST be set to "token".

client_id
REQUIRED. The client identifier as described in Section 2.2.

redirect_uri
OPTIONAL. As described in Section 3.1.2.

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

state
RECOMMENDED. An opaque value used by the client to maintain state between the request and callback. The authorization server includes this value when redirecting the user-agent back to the client. The parameter SHOULD be used for preventing cross-site request forgery as described in Section 10.12.

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

For example, the client directs the user-agent to make the following HTTP request using TLS (with extra line breaks for display purposes only):

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

The authorization server validates the request to ensure that all required parameters are present and valid. The authorization server MUST verify that the redirection URI to which it will redirect the access token matches a redirection URI registered by the client as described in Section 3.1.2.

If the request is valid, the authorization server authenticates the resource owner and obtains an authorization decision (by asking the resource owner or by establishing approval via other means).

When a decision is established, the authorization server directs the user-agent to the provided client redirection URI using an HTTP redirection response, or by other means available to it via the user-agent.

4.2.2. Access Token Response

If the resource owner grants the access request, the authorization server issues an access token and delivers it to the client by adding the following parameters to the fragment component of the redirection URI using the "application/x-www-form-urlencoded" format, per Appendix B:

access_token
: REQUIRED. The access token issued by the authorization server.

token_type
: REQUIRED. The type of the token issued as described in Section 7.1. Value is case insensitive.

expires_in
: RECOMMENDED. The lifetime in seconds of the access token. For example, the value "3600" denotes that the access token will expire in one hour from the time the response was generated. If omitted, the authorization server SHOULD provide the expiration time via other means or document the default value.

scope
: OPTIONAL, if identical to the scope requested by the client; otherwise, REQUIRED. The scope of the access token as described by Section 3.3.

state
: REQUIRED if the "state" parameter was present in the client authorization request. The exact value received from the client.

The authorization server MUST NOT issue a refresh token.

For example, the authorization server redirects the user-agent by sending the following HTTP response (with extra line breaks for display purposes only):

```
HTTP/1.1 302 Found
Location: http://example.com/cb#access_token=2YotnFZFEjr1zCsicMWpAA
          &state=xyz&token_type=example&expires_in=3600
```

Developers should note that some user-agents do not support the inclusion of a fragment component in the HTTP "Location" response header field. Such clients will require using other methods for redirecting the client than a 3xx redirection response -- for example, returning an HTML page that includes a 'continue' button with an action linked to the redirection URI.

The client MUST ignore unrecognized response parameters. The access token string size is left undefined by this specification. The client should avoid making assumptions about value sizes. The authorization server SHOULD document the size of any value it issues.

4.2.2.1. Error Response

If the request fails due to a missing, invalid, or mismatching redirection URI, or if the client identifier is missing or invalid, the authorization server SHOULD inform the resource owner of the error and MUST NOT automatically redirect the user-agent to the invalid redirection URI.

If the resource owner denies the access request or if the request fails for reasons other than a missing or invalid redirection URI, the authorization server informs the client by adding the following parameters to the fragment component of the redirection URI using the "application/x-www-form-urlencoded" format, per Appendix B:

error
: REQUIRED. A single ASCII [USASCII] error code from the following:

  invalid_request
  : The request is missing a required parameter, includes an invalid parameter value, includes a parameter more than once, or is otherwise malformed.

  unauthorized_client
  : The client is not authorized to request an access token using this method.

  access_denied
  : The resource owner or authorization server denied the request.

  unsupported_response_type
  : The authorization server does not support obtaining an access token using this method.

  invalid_scope
  : The requested scope is invalid, unknown, or malformed.

server_error
The authorization server encountered an unexpected condition that prevented it from fulfilling the request. (This error code is needed because a 500 Internal Server Error HTTP status code cannot be returned to the client via an HTTP redirect.)

temporarily_unavailable
The authorization server is currently unable to handle the request due to a temporary overloading or maintenance of the server. (This error code is needed because a 503 Service Unavailable HTTP status code cannot be returned to the client via an HTTP redirect.)

Values for the "error" parameter MUST NOT include characters outside the set %x20-21 / %x23-5B / %x5D-7E.

error_description
OPTIONAL. Human-readable ASCII [USASCII] text providing additional information, used to assist the client developer in understanding the error that occurred.
Values for the "error_description" parameter MUST NOT include characters outside the set %x20-21 / %x23-5B / %x5D-7E.

error_uri
OPTIONAL. A URI identifying a human-readable web page with information about the error, used to provide the client developer with additional information about the error.
Values for the "error_uri" parameter MUST conform to the URI-reference syntax and thus MUST NOT include characters outside the set %x21 / %x23-5B / %x5D-7E.

state
REQUIRED if a "state" parameter was present in the client authorization request. The exact value received from the client.

For example, the authorization server redirects the user-agent by sending the following HTTP response:

```
HTTP/1.1 302 Found
Location: https://client.example.com/cb#error=access_denied&state=xyz
```

4.3. Resource Owner Password Credentials Grant

The resource owner password credentials grant type is suitable in cases where the resource owner has a trust relationship with the client, such as the device operating system or a highly privileged

application. The authorization server should take special care when enabling this grant type and only allow it when other flows are not viable.

This grant type is suitable for clients capable of obtaining the resource owner's credentials (username and password, typically using an interactive form). It is also used to migrate existing clients using direct authentication schemes such as HTTP Basic or Digest authentication to OAuth by converting the stored credentials to an access token.

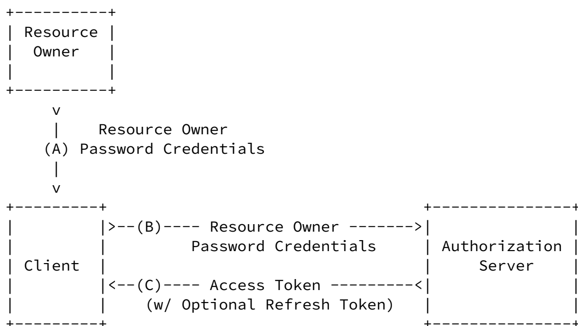


Figure 5: Resource Owner Password Credentials Flow

The flow illustrated in Figure 5 includes the following steps:

(A) The resource owner provides the client with its username and password.

(B) The client requests an access token from the authorization server's token endpoint by including the credentials received from the resource owner. When making the request, the client authenticates with the authorization server.

(C) The authorization server authenticates the client and validates the resource owner credentials, and if valid, issues an access token.

4.3.1. Authorization Request and Response

The method through which the client obtains the resource owner credentials is beyond the scope of this specification. The client MUST discard the credentials once an access token has been obtained.

4.3.2. Access Token Request

The client makes a request to the token endpoint by adding the following parameters using the "application/x-www-form-urlencoded" format per Appendix B with a character encoding of UTF-8 in the HTTP request entity-body:

grant_type
: REQUIRED. Value MUST be set to "password".

username
: REQUIRED. The resource owner username.

password
: REQUIRED. The resource owner password.

scope
: OPTIONAL. The scope of the access request as described by Section 3.3.

If the client type is confidential or the client was issued client credentials (or assigned other authentication requirements), the client MUST authenticate with the authorization server as described in Section 3.2.1.

For example, the client makes the following HTTP request using transport-layer security (with extra line breaks for display purposes only):

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

grant_type=password&username=johndoe&password=A3ddj3w
```

The authorization server MUST:

- o require client authentication for confidential clients or for any client that was issued client credentials (or with other authentication requirements),
- o authenticate the client if client authentication is included, and
- o validate the resource owner password credentials using its existing password validation algorithm.

Since this access token request utilizes the resource owner's password, the authorization server MUST protect the endpoint against brute force attacks (e.g., using rate-limitation or generating alerts).

4.3.3. Access Token Response

If the access token request is valid and authorized, the authorization server issues an access token and optional refresh token as described in Section 5.1. If the request failed client authentication or is invalid, the authorization server returns an error response as described in Section 5.2.

An example successful response:

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "access_token":"2YotnFZFEjr1zCsicMWpAA",
  "token_type":"example",
  "expires_in":3600,
  "refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
  "example_parameter":"example_value"
}
```

4.4. Client Credentials Grant

The client can request an access token using only its client credentials (or other supported means of authentication) when the client is requesting access to the protected resources under its control, or those of another resource owner that have been previously arranged with the authorization server (the method of which is beyond the scope of this specification).

The client credentials grant type MUST only be used by confidential clients.

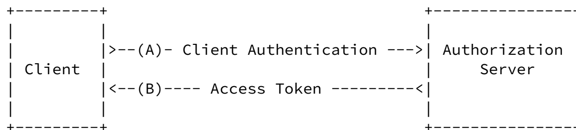


Figure 6: Client Credentials Flow

The flow illustrated in Figure 6 includes the following steps:

(A) The client authenticates with the authorization server and requests an access token from the token endpoint.

(B) The authorization server authenticates the client, and if valid, issues an access token.

4.4.1. Authorization Request and Response

Since the client authentication is used as the authorization grant, no additional authorization request is needed.

4.4.2. Access Token Request

The client makes a request to the token endpoint by adding the following parameters using the "application/x-www-form-urlencoded" format per Appendix B with a character encoding of UTF-8 in the HTTP request entity-body:

grant_type
: REQUIRED. Value MUST be set to "client_credentials".

scope
: OPTIONAL. The scope of the access request as described by Section 3.3.

The client MUST authenticate with the authorization server as described in Section 3.2.1.

For example, the client makes the following HTTP request using transport-layer security (with extra line breaks for display purposes only):

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

grant_type=client_credentials
```

The authorization server MUST authenticate the client.

4.4.3. Access Token Response

If the access token request is valid and authorized, the authorization server issues an access token as described in Section 5.1. A refresh token SHOULD NOT be included. If the request failed client authentication or is invalid, the authorization server returns an error response as described in Section 5.2.

An example successful response:

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "access_token":"2YotnFZFEjr1zCsicMWpAA",
  "token_type":"example",
  "expires_in":3600,
  "example_parameter":"example_value"
}
```

4.5. Extension Grants

The client uses an extension grant type by specifying the grant type using an absolute URI (defined by the authorization server) as the value of the "grant_type" parameter of the token endpoint, and by adding any additional parameters necessary.

For example, to request an access token using a Security Assertion Markup Language (SAML) 2.0 assertion grant type as defined by [OAuth-SAML2], the client could make the following HTTP request using TLS (with extra line breaks 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%3Asaml2-
bearer&assertion=PEFzc2VydGlvbiBJc3N1ZUluc3RhbnQ9IjIwMTEtMDU
[...omitted for brevity...]aG5TdGF0ZW1lbnQ-PC9Bc3NlcnRpb24-
```

If the access token request is valid and authorized, the authorization server issues an access token and optional refresh token as described in Section 5.1. If the request failed client authentication or is invalid, the authorization server returns an error response as described in Section 5.2.

5. Issuing an Access Token

If the access token request is valid and authorized, the authorization server issues an access token and optional refresh token as described in Section 5.1. If the request failed client authentication or is invalid, the authorization server returns an error response as described in Section 5.2.

5.1. Successful Response

The authorization server issues an access token and optional refresh token, and constructs the response by adding the following parameters to the entity-body of the HTTP response with a 200 (OK) status code:

access_token
: REQUIRED. The access token issued by the authorization server.

token_type
: REQUIRED. The type of the token issued as described in Section 7.1. Value is case insensitive.

expires_in
: RECOMMENDED. The lifetime in seconds of the access token. For example, the value "3600" denotes that the access token will expire in one hour from the time the response was generated. If omitted, the authorization server SHOULD provide the expiration time via other means or document the default value.

refresh_token
OPTIONAL. The refresh token, which can be used to obtain new access tokens using the same authorization grant as described in Section 6.

scope
OPTIONAL, if identical to the scope requested by the client; otherwise, REQUIRED. The scope of the access token as described by Section 3.3.

The parameters are included in the entity-body of the HTTP response using the "application/json" media type as defined by [RFC4627]. The parameters are serialized into a JavaScript Object Notation (JSON) structure by adding each parameter at the highest structure 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.

The authorization server MUST include the HTTP "Cache-Control" response header field [RFC2616] with a value of "no-store" in any response containing tokens, credentials, or other sensitive information, as well as the "Pragma" response header field [RFC2616] with a value of "no-cache".

For example:

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "access_token":"2YotnFZFEjr1zCsicMWpAA",
  "token_type":"example",
  "expires_in":3600,
  "refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
  "example_parameter":"example_value"
}
```

The client MUST ignore unrecognized value names in the response. The sizes of tokens and other values received from the authorization server are left undefined. The client should avoid making assumptions about value sizes. The authorization server SHOULD document the size of any value it issues.

5.2. Error Response

The authorization server responds with an HTTP 400 (Bad Request) status code (unless specified otherwise) and includes the following parameters with the response:

error
REQUIRED. A single ASCII [USASCII] error code from the following:

invalid_request
The request is missing a required parameter, includes an unsupported parameter value (other than grant type), repeats a parameter, includes multiple credentials, utilizes more than one mechanism for authenticating the client, or is otherwise malformed.

invalid_client
Client authentication failed (e.g., unknown client, no client authentication included, or unsupported authentication method). The authorization server MAY return an HTTP 401 (Unauthorized) status code to indicate which HTTP authentication schemes are supported. If the client attempted to authenticate via the "Authorization" request header field, the authorization server MUST respond with an HTTP 401 (Unauthorized) status code and include the "WWW-Authenticate" response header field matching the authentication scheme used by the client.

invalid_grant
The provided authorization grant (e.g., authorization code, resource owner credentials) or refresh token is invalid, expired, revoked, does not match the redirection URI used in the authorization request, or was issued to another client.

unauthorized_client
The authenticated client is not authorized to use this authorization grant type.

unsupported_grant_type
The authorization grant type is not supported by the authorization server.

invalid_scope
The requested scope is invalid, unknown, malformed, or exceeds the scope granted by the resource owner.

Values for the "error" parameter MUST NOT include characters outside the set %x20-21 / %x23-5B / %x5D-7E.

error_description
OPTIONAL. Human-readable ASCII [USASCII] text providing additional information, used to assist the client developer in understanding the error that occurred.
Values for the "error_description" parameter MUST NOT include characters outside the set %x20-21 / %x23-5B / %x5D-7E.

error_uri
OPTIONAL. A URI identifying a human-readable web page with information about the error, used to provide the client developer with additional information about the error.
Values for the "error_uri" parameter MUST conform to the URI-reference syntax and thus MUST NOT include characters outside the set %x21 / %x23-5B / %x5D-7E.

The parameters are included in the entity-body of the HTTP response using the "application/json" media type as defined by [RFC4627]. The parameters are serialized into a JSON structure by adding each parameter at the highest structure 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.

For example:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "error":"invalid_request"
}
```

6. Refreshing an Access Token

If the authorization server issued a refresh token to the client, the client makes a refresh request to the token endpoint by adding the following parameters using the "application/x-www-form-urlencoded" format per Appendix B with a character encoding of UTF-8 in the HTTP request entity-body:

grant_type
: REQUIRED. Value MUST be set to "refresh_token".

refresh_token
: REQUIRED. The refresh token issued to the client.

scope
: OPTIONAL. The scope of the access request as described by Section 3.3. The requested scope MUST NOT include any scope not originally granted by the resource owner, and if omitted is treated as equal to the scope originally granted by the resource owner.

Because refresh tokens are typically long-lasting credentials used to request additional access tokens, the refresh token is bound to the client to which it was issued. If the client type is confidential or the client was issued client credentials (or assigned other authentication requirements), the client MUST authenticate with the authorization server as described in Section 3.2.1.

For example, the client makes the following HTTP request using transport-layer security (with extra line breaks for display purposes only):

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

grant_type=refresh_token&refresh_token=tGzv3JOkF0XG5Qx2TlKWIA
```

The authorization server MUST:

- o require client authentication for confidential clients or for any client that was issued client credentials (or with other authentication requirements),
- o authenticate the client if client authentication is included and ensure that the refresh token was issued to the authenticated client, and
- o validate the refresh token.

If valid and authorized, the authorization server issues an access token as described in Section 5.1. If the request failed verification or is invalid, the authorization server returns an error response as described in Section 5.2.

The authorization server MAY issue a new refresh token, in which case the client MUST discard the old refresh token and replace it with the new refresh token. The authorization server MAY revoke the old refresh token after issuing a new refresh token to the client. If a new refresh token is issued, the refresh token scope MUST be identical to that of the refresh token included by the client in the request.

7. Accessing Protected Resources

The client accesses protected resources by presenting the access token to the resource server. The resource server MUST validate the access token and ensure that it has not expired and that its scope covers the requested resource. The methods used by the resource server to validate the access token (as well as any error responses) are beyond the scope of this specification but generally involve an interaction or coordination between the resource server and the authorization server.

The method in which the client utilizes the access token to authenticate with the resource server depends on the type of access token issued by the authorization server. Typically, it involves using the HTTP "Authorization" request header field [RFC2617] with an authentication scheme defined by the specification of the access token type used, such as [RFC6750].

7.1. Access Token Types

The access token type provides the client with the information required to successfully utilize the access token to make a protected resource request (along with type-specific attributes). The client MUST NOT use an access token if it does not understand the token type.

For example, the "bearer" token type defined in [RFC6750] is utilized by simply including the access token string in the request:

```
GET /resource/1 HTTP/1.1
Host: example.com
Authorization: Bearer mF_9.B5f-4.1JqM
```

while the "mac" token type defined in [OAuth-HTTP-MAC] is utilized by issuing a Message Authentication Code (MAC) key together with the access token that is used to sign certain components of the HTTP requests:

```
GET /resource/1 HTTP/1.1
Host: example.com
Authorization: MAC id="h480djs93hd8",
                   nonce="274312:dj83hs9s",
                   mac="kDZvddkndxvhGRXZhvuDjEWhGeE="
```

The above examples are provided for illustration purposes only. Developers are advised to consult the [RFC6750] and [OAuth-HTTP-MAC] specifications before use.

Each access token type definition specifies the additional attributes (if any) sent to the client together with the "access_token" response parameter. It also defines the HTTP authentication method used to include the access token when making a protected resource request.

7.2. Error Response

If a resource access request fails, the resource server SHOULD inform the client of the error. While the specifics of such error responses are beyond the scope of this specification, this document establishes a common registry in Section 11.4 for error values to be shared among OAuth token authentication schemes.

New authentication schemes designed primarily for OAuth token authentication SHOULD define a mechanism for providing an error status code to the client, in which the error values allowed are registered in the error registry established by this specification.

Such schemes MAY limit the set of valid error codes to a subset of the registered values. If the error code is returned using a named parameter, the parameter name SHOULD be "error".

Other schemes capable of being used for OAuth token authentication, but not primarily designed for that purpose, MAY bind their error values to the registry in the same manner.

New authentication schemes MAY choose to also specify the use of the "error_description" and "error_uri" parameters to return error information in a manner parallel to their usage in this specification.

8. Extensibility

8.1. Defining Access Token Types

Access token types can be defined in one of two ways: registered in the Access Token Types registry (following the procedures in Section 11.1), or by using a unique absolute URI as its name.

Types utilizing a URI name SHOULD be limited to vendor-specific implementations that are not commonly applicable, and are specific to the implementation details of the resource server where they are used.

All other types MUST be registered. Type names MUST conform to the type-name ABNF. If the type definition includes a new HTTP authentication scheme, the type name SHOULD be identical to the HTTP authentication scheme name (as defined by [RFC2617]). The token type "example" is reserved for use in examples.

```
type-name  = 1*name-char
name-char  = "-" / "." / "_" / DIGIT / ALPHA
```

8.2. Defining New Endpoint Parameters

New request or response parameters for use with the authorization endpoint or the token endpoint are defined and registered in the OAuth Parameters registry following the procedure in Section 11.2.

Parameter names MUST conform to the param-name ABNF, and parameter values syntax MUST be well-defined (e.g., using ABNF, or a reference to the syntax of an existing parameter).

```
param-name  = 1*name-char
name-char   = "-" / "." / "_" / DIGIT / ALPHA
```

Unregistered vendor-specific parameter extensions that are not commonly applicable and that are specific to the implementation details of the authorization server where they are used SHOULD utilize a vendor-specific prefix that is not likely to conflict with other registered values (e.g., begin with 'companyname_').

8.3. Defining New Authorization Grant Types

New authorization grant types can be defined by assigning them a unique absolute URI for use with the "grant_type" parameter. If the extension grant type requires additional token endpoint parameters, they MUST be registered in the OAuth Parameters registry as described by Section 11.2.

8.4. Defining New Authorization Endpoint Response Types

New response types for use with the authorization endpoint are defined and registered in the Authorization Endpoint Response Types registry following the procedure in Section 11.3. Response type names MUST conform to the response-type ABNF.

```
response-type  = response-name *( SP response-name )
response-name  = 1*response-char
response-char  = "_" / DIGIT / ALPHA
```

If a response type contains one or more space characters (%x20), it is compared as a space-delimited list of values in which the order of values does not matter. Only one order of values can be registered, which covers all other arrangements of the same set of values.

For example, the response type "token code" is left undefined by this specification. However, an extension can define and register the "token code" response type. Once registered, the same combination cannot be registered as "code token", but both values can be used to denote the same response type.

8.5. Defining Additional Error Codes

In cases where protocol extensions (i.e., access token types, extension parameters, or extension grant types) require additional error codes to be used with the authorization code grant error response (Section 4.1.2.1), the implicit grant error response (Section 4.2.2.1), the token error response (Section 5.2), or the resource access error response (Section 7.2), such error codes MAY be defined.

Extension error codes MUST be registered (following the procedures in Section 11.4) if the extension they are used in conjunction with is a registered access token type, a registered endpoint parameter, or an extension grant type. Error codes used with unregistered extensions MAY be registered.

Error codes MUST conform to the error ABNF and SHOULD be prefixed by an identifying name when possible. For example, an error identifying an invalid value set to the extension parameter "example" SHOULD be named "example_invalid".

```
error      = 1*error-char
error-char = %x20-21 / %x23-5B / %x5D-7E
```

9. Native Applications

Native applications are clients installed and executed on the device used by the resource owner (i.e., desktop application, native mobile application). Native applications require special consideration related to security, platform capabilities, and overall end-user experience.

The authorization endpoint requires interaction between the client and the resource owner's user-agent. Native applications can invoke an external user-agent or embed a user-agent within the application. For example:

- External user-agent - the native application can capture the response from the authorization server using a redirection URI with a scheme registered with the operating system to invoke the client as the handler, manual copy-and-paste of the credentials, running a local web server, installing a user-agent extension, or by providing a redirection URI identifying a server-hosted resource under the client's control, which in turn makes the response available to the native application.

- Embedded user-agent - the native application obtains the response by directly communicating with the embedded user-agent by monitoring state changes emitted during the resource load, or accessing the user-agent's cookies storage.

When choosing between an external or embedded user-agent, developers should consider the following:

- An external user-agent may improve completion rate, as the resource owner may already have an active session with the authorization server, removing the need to re-authenticate. It provides a familiar end-user experience and functionality. The

resource owner may also rely on user-agent features or extensions to assist with authentication (e.g., password manager, 2-factor device reader).

- An embedded user-agent may offer improved usability, as it removes the need to switch context and open new windows.

- An embedded user-agent poses a security challenge because resource owners are authenticating in an unidentified window without access to the visual protections found in most external user-agents. An embedded user-agent educates end-users to trust unidentified requests for authentication (making phishing attacks easier to execute).

When choosing between the implicit grant type and the authorization code grant type, the following should be considered:

- Native applications that use the authorization code grant type SHOULD do so without using client credentials, due to the native application's inability to keep client credentials confidential.

- When using the implicit grant type flow, a refresh token is not returned, which requires repeating the authorization process once the access token expires.

10. Security Considerations

As a flexible and extensible framework, OAuth's security considerations depend on many factors. The following sections provide implementers with security guidelines focused on the three client profiles described in Section 2.1: web application, user-agent-based application, and native application.

A comprehensive OAuth security model and analysis, as well as background for the protocol design, is provided by [OAuth-THREATMODEL].

10.1. Client Authentication

The authorization server establishes client credentials with web application clients for the purpose of client authentication. The authorization server is encouraged to consider stronger client authentication means than a client password. Web application clients MUST ensure confidentiality of client passwords and other client credentials.

The authorization server MUST NOT issue client passwords or other client credentials to native application or user-agent-based application clients for the purpose of client authentication. The authorization server MAY issue a client password or other credentials for a specific installation of a native application client on a specific device.

When client authentication is not possible, the authorization server SHOULD employ other means to validate the client's identity -- for example, by requiring the registration of the client redirection URI or enlisting the resource owner to confirm identity. A valid redirection URI is not sufficient to verify the client's identity when asking for resource owner authorization but can be used to prevent delivering credentials to a counterfeit client after obtaining resource owner authorization.

The authorization server must consider the security implications of interacting with unauthenticated clients and take measures to limit the potential exposure of other credentials (e.g., refresh tokens) issued to such clients.

10.2. Client Impersonation

A malicious client can impersonate another client and obtain access to protected resources if the impersonated client fails to, or is unable to, keep its client credentials confidential.

The authorization server MUST authenticate the client whenever possible. If the authorization server cannot authenticate the client due to the client's nature, the authorization server MUST require the registration of any redirection URI used for receiving authorization responses and SHOULD utilize other means to protect resource owners from such potentially malicious clients. For example, the authorization server can engage the resource owner to assist in identifying the client and its origin.

The authorization server SHOULD enforce explicit resource owner authentication and provide the resource owner with information about the client and the requested authorization scope and lifetime. It is up to the resource owner to review the information in the context of the current client and to authorize or deny the request.

The authorization server SHOULD NOT process repeated authorization requests automatically (without active resource owner interaction) without authenticating the client or relying on other measures to ensure that the repeated request comes from the original client and not an impersonator.

10.3. Access Tokens

Access token credentials (as well as any confidential access token attributes) MUST be kept confidential in transit and storage, and only shared among the authorization server, the resource servers the access token is valid for, and the client to whom the access token is issued. Access token credentials MUST only be transmitted using TLS as described in Section 1.6 with server authentication as defined by [RFC2818].

When using the implicit grant type, the access token is transmitted in the URI fragment, which can expose it to unauthorized parties.

The authorization server MUST ensure that access tokens cannot be generated, modified, or guessed to produce valid access tokens by unauthorized parties.

The client SHOULD request access tokens with the minimal scope necessary. The authorization server SHOULD take the client identity into account when choosing how to honor the requested scope and MAY issue an access token with less rights than requested.

This specification does not provide any methods for the resource server to ensure that an access token presented to it by a given client was issued to that client by the authorization server.

10.4. Refresh Tokens

Authorization servers MAY issue refresh tokens to web application clients and native application clients.

Refresh tokens MUST be kept confidential in transit and storage, and shared only among the authorization server and the client to whom the refresh tokens were issued. The authorization server MUST maintain the binding between a refresh token and the client to whom it was issued. Refresh tokens MUST only be transmitted using TLS as described in Section 1.6 with server authentication as defined by [RFC2818].

The authorization server MUST verify the binding between the refresh token and client identity whenever the client identity can be authenticated. When client authentication is not possible, the authorization server SHOULD deploy other means to detect refresh token abuse.

For example, the authorization server could employ refresh token rotation in which a new refresh token is issued with every access token refresh response. The previous refresh token is invalidated

but retained by the authorization server. If a refresh token is compromised and subsequently used by both the attacker and the legitimate client, one of them will present an invalidated refresh token, which will inform the authorization server of the breach.

The authorization server MUST ensure that refresh tokens cannot be generated, modified, or guessed to produce valid refresh tokens by unauthorized parties.

10.5. Authorization Codes

The transmission of authorization codes SHOULD be made over a secure channel, and the client SHOULD require the use of TLS with its redirection URI if the URI identifies a network resource. Since authorization codes are transmitted via user-agent redirections, they could potentially be disclosed through user-agent history and HTTP referrer headers.

Authorization codes operate as plaintext bearer credentials, used to verify that the resource owner who granted authorization at the authorization server is the same resource owner returning to the client to complete the process. Therefore, if the client relies on the authorization code for its own resource owner authentication, the client redirection endpoint MUST require the use of TLS.

Authorization codes MUST be short lived and single-use. If the authorization server observes multiple attempts to exchange an authorization code for an access token, the authorization server SHOULD attempt to revoke all access tokens already granted based on the compromised authorization code.

If the client can be authenticated, the authorization servers MUST authenticate the client and ensure that the authorization code was issued to the same client.

10.6. Authorization Code Redirection URI Manipulation

When requesting authorization using the authorization code grant type, the client can specify a redirection URI via the "redirect_uri" parameter. If an attacker can manipulate the value of the redirection URI, it can cause the authorization server to redirect the resource owner user-agent to a URI under the control of the attacker with the authorization code.

An attacker can create an account at a legitimate client and initiate the authorization flow. When the attacker's user-agent is sent to the authorization server to grant access, the attacker grabs the authorization URI provided by the legitimate client and replaces the

client's redirection URI with a URI under the control of the attacker. The attacker then tricks the victim into following the manipulated link to authorize access to the legitimate client.

Once at the authorization server, the victim is prompted with a normal, valid request on behalf of a legitimate and trusted client, and authorizes the request. The victim is then redirected to an endpoint under the control of the attacker with the authorization code. The attacker completes the authorization flow by sending the authorization code to the client using the original redirection URI provided by the client. The client exchanges the authorization code with an access token and links it to the attacker's client account, which can now gain access to the protected resources authorized by the victim (via the client).

In order to prevent such an attack, the authorization server MUST ensure that the redirection URI used to obtain the authorization code is identical to the redirection URI provided when exchanging the authorization code for an access token. The authorization server MUST require public clients and SHOULD require confidential clients to register their redirection URIs. If a redirection URI is provided in the request, the authorization server MUST validate it against the registered value.

10.7. Resource Owner Password Credentials

The resource owner password credentials grant type is often used for legacy or migration reasons. It reduces the overall risk of storing usernames and passwords by the client but does not eliminate the need to expose highly privileged credentials to the client.

This grant type carries a higher risk than other grant types because it maintains the password anti-pattern this protocol seeks to avoid. The client could abuse the password, or the password could unintentionally be disclosed to an attacker (e.g., via log files or other records kept by the client).

Additionally, because the resource owner does not have control over the authorization process (the resource owner's involvement ends when it hands over its credentials to the client), the client can obtain access tokens with a broader scope than desired by the resource owner. The authorization server should consider the scope and lifetime of access tokens issued via this grant type.

The authorization server and client SHOULD minimize use of this grant type and utilize other grant types whenever possible.

10.8. Request Confidentiality

Access tokens, refresh tokens, resource owner passwords, and client credentials MUST NOT be transmitted in the clear. Authorization codes SHOULD NOT be transmitted in the clear.

The "state" and "scope" parameters SHOULD NOT include sensitive client or resource owner information in plain text, as they can be transmitted over insecure channels or stored insecurely.

10.9. Ensuring Endpoint Authenticity

In order to prevent man-in-the-middle attacks, the authorization server MUST require the use of TLS with server authentication as defined by [RFC2818] for any request sent to the authorization and token endpoints. The client MUST validate the authorization server's TLS certificate as defined by [RFC6125] and in accordance with its requirements for server identity authentication.

10.10. Credentials-Guessing Attacks

The authorization server MUST prevent attackers from guessing access tokens, authorization codes, refresh tokens, resource owner passwords, and client credentials.

The probability of an attacker guessing generated tokens (and other credentials not intended for handling by end-users) MUST be less than or equal to 2^(-128) and SHOULD be less than or equal to 2^(-160).

The authorization server MUST utilize other means to protect credentials intended for end-user usage.

10.11. Phishing Attacks

Wide deployment of this and similar protocols may cause end-users to become inured to the practice of being redirected to websites where they are asked to enter their passwords. If end-users are not careful to verify the authenticity of these websites before entering their credentials, it will be possible for attackers to exploit this practice to steal resource owners' passwords.

Service providers should attempt to educate end-users about the risks phishing attacks pose and should provide mechanisms that make it easy for end-users to confirm the authenticity of their sites. Client developers should consider the security implications of how they interact with the user-agent (e.g., external, embedded), and the ability of the end-user to verify the authenticity of the authorization server.

To reduce the risk of phishing attacks, the authorization servers MUST require the use of TLS on every endpoint used for end-user interaction.

10.12. Cross-Site Request Forgery

Cross-site request forgery (CSRF) is an exploit in which an attacker causes the user-agent of a victim end-user to follow a malicious URI (e.g., provided to the user-agent as a misleading link, image, or redirection) to a trusting server (usually established via the presence of a valid session cookie).

A CSRF attack against the client's redirection URI allows an attacker to inject its own authorization code or access token, which can result in the client using an access token associated with the attacker's protected resources rather than the victim's (e.g., save the victim's bank account information to a protected resource controlled by the attacker).

The client MUST implement CSRF protection for its redirection URI. This is typically accomplished by requiring any request sent to the redirection URI endpoint to include a value that binds the request to the user-agent's authenticated state (e.g., a hash of the session cookie used to authenticate the user-agent). The client SHOULD utilize the "state" request parameter to deliver this value to the authorization server when making an authorization request.

Once authorization has been obtained from the end-user, the authorization server redirects the end-user's user-agent back to the client with the required binding value contained in the "state" parameter. The binding value enables the client to verify the validity of the request by matching the binding value to the user-agent's authenticated state. The binding value used for CSRF protection MUST contain a non-guessable value (as described in Section 10.10), and the user-agent's authenticated state (e.g., session cookie, HTML5 local storage) MUST be kept in a location accessible only to the client and the user-agent (i.e., protected by same-origin policy).

A CSRF attack against the authorization server's authorization endpoint can result in an attacker obtaining end-user authorization for a malicious client without involving or alerting the end-user.

The authorization server MUST implement CSRF protection for its authorization endpoint and ensure that a malicious client cannot obtain authorization without the awareness and explicit consent of the resource owner.

10.13. Clickjacking

In a clickjacking attack, an attacker registers a legitimate client and then constructs a malicious site in which it loads the authorization server's authorization endpoint web page in a transparent iframe overlaid on top of a set of dummy buttons, which are carefully constructed to be placed directly under important buttons on the authorization page. When an end-user clicks a misleading visible button, the end-user is actually clicking an invisible button on the authorization page (such as an "Authorize" button). This allows an attacker to trick a resource owner into granting its client access without the end-user's knowledge.

To prevent this form of attack, native applications SHOULD use external browsers instead of embedding browsers within the application when requesting end-user authorization. For most newer browsers, avoidance of iframes can be enforced by the authorization server using the (non-standard) "x-frame-options" header. This header can have two values, "deny" and "sameorigin", which will block any framing, or framing by sites with a different origin, respectively. For older browsers, JavaScript frame-busting techniques can be used but may not be effective in all browsers.

10.14. Code Injection and Input Validation

A code injection attack occurs when an input or otherwise external variable is used by an application unsanitized and causes modification to the application logic. This may allow an attacker to gain access to the application device or its data, cause denial of service, or introduce a wide range of malicious side-effects.

The authorization server and client MUST sanitize (and validate when possible) any value received -- in particular, the value of the "state" and "redirect_uri" parameters.

10.15. Open Redirectors

The authorization server, authorization endpoint, and client redirection endpoint can be improperly configured and operate as open redirectors. An open redirector is an endpoint using a parameter to automatically redirect a user-agent to the location specified by the parameter value without any validation.

Open redirectors can be used in phishing attacks, or by an attacker to get end-users to visit malicious sites by using the URI authority component of a familiar and trusted destination. In addition, if the authorization server allows the client to register only part of the redirection URI, an attacker can use an open redirector operated by

the client to construct a redirection URI that will pass the authorization server validation but will send the authorization code or access token to an endpoint under the control of the attacker.

10.16. Misuse of Access Token to Impersonate Resource Owner in Implicit Flow

For public clients using implicit flows, this specification does not provide any method for the client to determine what client an access token was issued to.

A resource owner may willingly delegate access to a resource by granting an access token to an attacker's malicious client. This may be due to phishing or some other pretext. An attacker may also steal a token via some other mechanism. An attacker may then attempt to impersonate the resource owner by providing the access token to a legitimate public client.

In the implicit flow (response_type=token), the attacker can easily switch the token in the response from the authorization server, replacing the real access token with the one previously issued to the attacker.

Servers communicating with native applications that rely on being passed an access token in the back channel to identify the user of the client may be similarly compromised by an attacker creating a compromised application that can inject arbitrary stolen access tokens.

Any public client that makes the assumption that only the resource owner can present it with a valid access token for the resource is vulnerable to this type of attack.

This type of attack may expose information about the resource owner at the legitimate client to the attacker (malicious client). This will also allow the attacker to perform operations at the legitimate client with the same permissions as the resource owner who originally granted the access token or authorization code.

Authenticating resource owners to clients is out of scope for this specification. Any specification that uses the authorization process as a form of delegated end-user authentication to the client (e.g., third-party sign-in service) MUST NOT use the implicit flow without additional security mechanisms that would enable the client to determine if the access token was issued for its use (e.g., audience-restricting the access token).

11. IANA Considerations

11.1. OAuth Access Token Types Registry

This specification establishes the OAuth Access Token Types registry.

Access token types are registered with a Specification Required ([RFC5226]) after a two-week review period on the oauth-ext-review@ietf.org mailing list, on the advice of one or more Designated Experts. However, to allow for the allocation of values prior to publication, the Designated Expert(s) may approve registration once they are satisfied that such a specification will be published.

Registration requests must be sent to the oauth-ext-review@ietf.org mailing list for review and comment, with an appropriate subject (e.g., "Request for access token type: example").

Within the review period, the Designated Expert(s) will either approve or deny the registration request, communicating this decision to the review list and IANA. Denials should include an explanation and, if applicable, suggestions as to how to make the request successful.

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

11.1.1. Registration Template

Type name:
The name requested (e.g., "example").

Additional Token Endpoint Response Parameters:
Additional response parameters returned together with the "access_token" parameter. New parameters MUST be separately registered in the OAuth Parameters registry as described by Section 11.2.

HTTP Authentication Scheme(s):
The HTTP authentication scheme name(s), if any, used to authenticate protected resource requests using access tokens of this type.

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

Specification document(s):
Reference to the document(s) that specify the parameter, preferably including a URI that can be used to retrieve a copy of the document(s). An indication of the relevant sections may also be included but is not required.

11.2. OAuth Parameters Registry

This specification establishes the OAuth Parameters registry.

Additional parameters for inclusion in the authorization endpoint request, the authorization endpoint response, the token endpoint request, or the token endpoint response are registered with a Specification Required ([RFC5226]) after a two-week review period on the oauth-ext-review@ietf.org mailing list, on the advice of one or more Designated Experts. However, to allow for the allocation of values prior to publication, the Designated Expert(s) may approve registration once they are satisfied that such a specification will be published.

Registration requests must be sent to the oauth-ext-review@ietf.org mailing list for review and comment, with an appropriate subject (e.g., "Request for parameter: example").

Within the review period, the Designated Expert(s) will either approve or deny the registration request, communicating this decision to the review list and IANA. Denials should include an explanation and, if applicable, suggestions as to how to make the request successful.

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

11.2.1. Registration Template

Parameter name:
The name requested (e.g., "example").

Parameter usage location:
The location(s) where parameter can be used. The possible locations are authorization request, authorization response, token request, or token response.

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

Specification document(s):
Reference to the document(s) that specify the parameter, preferably including a URI that can be used to retrieve a copy of the document(s). An indication of the relevant sections may also be included but is not required.

11.2.2. Initial Registry Contents

The OAuth Parameters registry's initial contents are:

- Parameter name: client_id
- Parameter usage location: authorization request, token request
- Change controller: IETF
- Specification document(s): RFC 6749

- Parameter name: client_secret
- Parameter usage location: token request
- Change controller: IETF
- Specification document(s): RFC 6749

- Parameter name: response_type
- Parameter usage location: authorization request
- Change controller: IETF
- Specification document(s): RFC 6749

- Parameter name: redirect_uri
- Parameter usage location: authorization request, token request
- Change controller: IETF
- Specification document(s): RFC 6749

- Parameter name: scope
- Parameter usage location: authorization request, authorization response, token request, token response
- Change controller: IETF
- Specification document(s): RFC 6749

- Parameter name: state
- Parameter usage location: authorization request, authorization response
- Change controller: IETF
- Specification document(s): RFC 6749

- Parameter name: code
- Parameter usage location: authorization response, token request
- Change controller: IETF
- Specification document(s): RFC 6749

- Parameter name: error_description
- Parameter usage location: authorization response, token response
- Change controller: IETF
- Specification document(s): RFC 6749

- Parameter name: error_uri
- Parameter usage location: authorization response, token response
- Change controller: IETF
- Specification document(s): RFC 6749

- Parameter name: grant_type
- Parameter usage location: token request
- Change controller: IETF
- Specification document(s): RFC 6749

- Parameter name: access_token
- Parameter usage location: authorization response, token response
- Change controller: IETF
- Specification document(s): RFC 6749

- Parameter name: token_type
- Parameter usage location: authorization response, token response
- Change controller: IETF
- Specification document(s): RFC 6749

- Parameter name: expires_in
- Parameter usage location: authorization response, token response
- Change controller: IETF
- Specification document(s): RFC 6749

- Parameter name: username
- Parameter usage location: token request
- Change controller: IETF
- Specification document(s): RFC 6749

- Parameter name: password
- Parameter usage location: token request
- Change controller: IETF
- Specification document(s): RFC 6749

- Parameter name: refresh_token
- Parameter usage location: token request, token response
- Change controller: IETF
- Specification document(s): RFC 6749

11.3. OAuth Authorization Endpoint Response Types Registry

This specification establishes the OAuth Authorization Endpoint Response Types registry.

Additional response types for use with the authorization endpoint are registered with a Specification Required ([RFC5226]) after a two-week review period on the oauth-ext-review@ietf.org mailing list, on the advice of one or more Designated Experts. However, to allow for the allocation of values prior to publication, the Designated Expert(s) may approve registration once they are satisfied that such a specification will be published.

Registration requests must be sent to the oauth-ext-review@ietf.org mailing list for review and comment, with an appropriate subject (e.g., "Request for response type: example").

Within the review period, the Designated Expert(s) will either approve or deny the registration request, communicating this decision to the review list and IANA. Denials should include an explanation and, if applicable, suggestions as to how to make the request successful.

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

11.3.1. Registration Template

Response type name:
The name requested (e.g., "example").

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

Specification document(s):
Reference to the document(s) that specify the type, preferably including a URI that can be used to retrieve a copy of the document(s). An indication of the relevant sections may also be included but is not required.

11.3.2. Initial Registry Contents

The OAuth Authorization Endpoint Response Types registry's initial contents are:

- Response type name: code
- Change controller: IETF
- Specification document(s): RFC 6749

- Response type name: token
- Change controller: IETF
- Specification document(s): RFC 6749

11.4. OAuth Extensions Error Registry

This specification establishes the OAuth Extensions Error registry.

Additional error codes used together with other protocol extensions (i.e., extension grant types, access token types, or extension parameters) are registered with a Specification Required ([RFC5226]) after a two-week review period on the oauth-ext-review@ietf.org mailing list, on the advice of one or more Designated Experts. However, to allow for the allocation of values prior to publication, the Designated Expert(s) may approve registration once they are satisfied that such a specification will be published.

Registration requests must be sent to the oauth-ext-review@ietf.org mailing list for review and comment, with an appropriate subject (e.g., "Request for error code: example").

Within the review period, the Designated Expert(s) will either approve or deny the registration request, communicating this decision to the review list and IANA. Denials should include an explanation and, if applicable, suggestions as to how to make the request successful.

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

11.4.1. Registration Template

Error name:
The name requested (e.g., "example"). Values for the error name MUST NOT include characters outside the set %x20-21 / %x23-5B / %x5D-7E.

Error usage location:
The location(s) where the error can be used. The possible locations are authorization code grant error response (Section 4.1.2.1), implicit grant error response (Section 4.2.2.1), token error response (Section 5.2), or resource access error response (Section 7.2).

Related protocol extension:
The name of the extension grant type, access token type, or extension parameter that the error code is used in conjunction with.

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

Specification document(s):
Reference to the document(s) that specify the error code, preferably including a URI that can be used to retrieve a copy of the document(s). An indication of the relevant sections may also be included but is not required.

12. References

12.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2246] Dierks, T. and C. Allen, "The TLS Protocol Version 1.0", RFC 2246, January 1999.

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[RFC2617] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and L. Stewart, "HTTP Authentication: Basic and Digest Access Authentication", RFC 2617, June 1999.

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.

[RFC4627] Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)", RFC 4627, July 2006.

[RFC4949] Shirey, R., "Internet Security Glossary, Version 2", RFC 4949, August 2007.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.

[RFC5234] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

[RFC6125] Saint-Andre, P. and J. Hodges, "Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS)", RFC 6125, March 2011.

[USASCII] American National Standards Institute, "Coded Character Set -- 7-bit American Standard Code for Information Interchange", ANSI X3.4, 1986.

[W3C.REC-html401-19991224]
Raggett, D., Le Hors, A., and I. Jacobs, "HTML 4.01 Specification", World Wide Web Consortium Recommendation REC-html401-19991224, December 1999, .

[W3C.REC-xml-20081126]
Bray, T., Paoli, J., Sperberg-McQueen, C., Maler, E., and F. Yergeau, "Extensible Markup Language (XML) 1.0 (Fifth Edition)", World Wide Web Consortium Recommendation REC-xml-20081126, November 2008, .

12.2. Informative References

[OAuth-HTTP-MAC]
Hammer-Lahav, E., Ed., "HTTP Authentication: MAC Access Authentication", Work in Progress, February 2012.

[OAuth-SAML2]
Campbell, B. and C. Mortimore, "SAML 2.0 Bearer Assertion Profiles for OAuth 2.0", Work in Progress, September 2012.

[OAuth-THREATMODEL]
Lodderstedt, T., Ed., McGloin, M., and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations", Work in Progress, October 2012.

[OAuth-WRAP]
Hardt, D., Ed., Tom, A., Eaton, B., and Y. Goland, "OAuth Web Resource Authorization Profiles", Work in Progress, January 2010.

[RFC5849] Hammer-Lahav, E., "The OAuth 1.0 Protocol", RFC 5849, April 2010.

[RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, October 2012.

Appendix A. Augmented Backus-Naur Form (ABNF) Syntax

This section provides Augmented Backus-Naur Form (ABNF) syntax descriptions for the elements defined in this specification using the notation of [RFC5234]. The ABNF below is defined in terms of Unicode code points [W3C.REC-xml-20081126]; these characters are typically encoded in UTF-8. Elements are presented in the order first defined.

Some of the definitions that follow use the "URI-reference" definition from [RFC3986].

Some of the definitions that follow use these common definitions:

```
VSCHAR     = %x20-7E
NQCHAR     = %x21 / %x23-5B / %x5D-7E
NQSCHAR    = %x20-21 / %x23-5B / %x5D-7E
UNICODECHARNOCRLF = %x09 /%x20-7E / %x80-D7FF /
                    %xE000-FFFD / %x10000-10FFFF
```

(The UNICODECHARNOCRLF definition is based upon the Char definition in Section 2.2 of [W3C.REC-xml-20081126], but omitting the Carriage Return and Linefeed characters.)

A.1. "client_id" Syntax

The "client_id" element is defined in Section 2.3.1:

```
client-id     = *VSCHAR
```

A.2. "client_secret" Syntax

The "client_secret" element is defined in Section 2.3.1:

```
client-secret = *VSCHAR
```

A.3. "response_type" Syntax

The "response_type" element is defined in Sections 3.1.1 and 8.4:

```
response-type = response-name *( SP response-name )
response-name = 1*response-char
response-char = "_" / DIGIT / ALPHA
```

A.4. "scope" Syntax

The "scope" element is defined in Section 3.3:

```
scope       = scope-token *( SP scope-token )
scope-token = 1*NQCHAR
```

A.5. "state" Syntax

The "state" element is defined in Sections 4.1.1, 4.1.2, 4.1.2.1, 4.2.1, 4.2.2, and 4.2.2.1:

```
state      = 1*VSCHAR
```

A.6. "redirect_uri" Syntax

The "redirect_uri" element is defined in Sections 4.1.1, 4.1.3, and 4.2.1:

```
redirect-uri      = URI-reference
```

A.7. "error" Syntax

The "error" element is defined in Sections 4.1.2.1, 4.2.2.1, 5.2, 7.2, and 8.5:

```
error             = 1*NQSCHAR
```

A.8. "error_description" Syntax

The "error_description" element is defined in Sections 4.1.2.1, 4.2.2.1, 5.2, and 7.2:

```
error-description = 1*NQSCHAR
```

A.9. "error_uri" Syntax

The "error_uri" element is defined in Sections 4.1.2.1, 4.2.2.1, 5.2, and 7.2:

```
error-uri         = URI-reference
```

A.10. "grant_type" Syntax

The "grant_type" element is defined in Sections 4.1.3, 4.3.2, 4.4.2, 4.5, and 6:

```
grant-type = grant-name / URI-reference
grant-name = 1*name-char
name-char  = "-" / "." / "_" / DIGIT / ALPHA
```

A.11. "code" Syntax

The "code" element is defined in Section 4.1.3:

```
code       = 1*VSCHAR
```

A.12. "access_token" Syntax

The "access_token" element is defined in Sections 4.2.2 and 5.1:

```
access-token = 1*VSCHAR
```

A.13. "token_type" Syntax

The "token_type" element is defined in Sections 4.2.2, 5.1, and 8.1:

```
token-type = type-name / URI-reference
type-name  = 1*name-char
name-char  = "-" / "." / "_" / DIGIT / ALPHA
```

A.14. "expires_in" Syntax

The "expires_in" element is defined in Sections 4.2.2 and 5.1:

```
expires-in = 1*DIGIT
```

A.15. "username" Syntax

The "username" element is defined in Section 4.3.2:

```
username = *UNICODECHARNOCRLF
```

A.16. "password" Syntax

The "password" element is defined in Section 4.3.2:

```
password = *UNICODECHARNOCRLF
```

A.17. "refresh_token" Syntax

The "refresh_token" element is defined in Sections 5.1 and 6:

```
refresh-token = 1*VSCHAR
```

A.18. Endpoint Parameter Syntax

The syntax for new endpoint parameters is defined in Section 8.2:

```
param-name = 1*name-char
name-char  = "-" / "." / "_" / DIGIT / ALPHA
```

Appendix B. Use of application/x-www-form-urlencoded Media Type

At the time of publication of this specification, the "application/x-www-form-urlencoded" media type was defined in Section 17.13.4 of [W3C.REC-html401-19991224] but not registered in the IANA MIME Media Types registry (). Furthermore, that definition is incomplete, as it does not consider non-US-ASCII characters.

To address this shortcoming when generating payloads using this media type, names and values MUST be encoded using the UTF-8 character encoding scheme [RFC3629] first; the resulting octet sequence then needs to be further encoded using the escaping rules defined in [W3C.REC-html401-19991224].

When parsing data from a payload using this media type, the names and values resulting from reversing the name/value encoding consequently need to be treated as octet sequences, to be decoded using the UTF-8 character encoding scheme.

For example, the value consisting of the six Unicode code points (1) U+0020 (SPACE), (2) U+0025 (PERCENT SIGN), (3) U+0026 (AMPERSAND), (4) U+002B (PLUS SIGN), (5) U+00A3 (POUND SIGN), and (6) U+20AC (EURO SIGN) would be encoded into the octet sequence below (using hexadecimal notation):

```
20 25 26 2B C2 A3 E2 82 AC
```

and then represented in the payload as:

```
+%25%26%2B%C2%A3%E2%82%AC
```

Appendix C. Acknowledgements

The initial OAuth 2.0 protocol specification was edited by David Recordon, based on two previous publications: the OAuth 1.0 community specification [RFC5849], and OAuth WRAP (OAuth Web Resource Authorization Profiles) [OAuth-WRAP]. Eran Hammer then edited many of the intermediate drafts that evolved into this RFC. The Security Considerations section was drafted by Torsten Lodderstedt, Mark McGloin, Phil Hunt, Anthony Nadalin, and John Bradley. The section on use of the "application/x-www-form-urlencoded" media type was drafted by Julian Reschke. The ABNF section was drafted by Michael B. Jones.

The OAuth 1.0 community specification was edited by Eran Hammer and authored by Mark Atwood, Dirk Balfanz, Darren Bounds, Richard M. Conlan, Blaine Cook, Leah Culver, Breno de Medeiros, Brian Eaton, Kellan Elliott-McCrea, Larry Halff, Eran Hammer, Ben Laurie, Chris Messina, John Panzer, Sam Quigley, David Recordon, Eran Sandler, Jonathan Sergent, Todd Sieling, Brian Slesinsky, and Andy Smith.

The OAuth WRAP specification was edited by Dick Hardt and authored by Brian Eaton, Yaron Y. Goland, Dick Hardt, and Allen Tom.

This specification is the work of the OAuth Working Group, which includes dozens of active and dedicated participants. In particular, the following individuals contributed ideas, feedback, and wording that shaped and formed the final specification:

Michael Adams, Amanda Anganes, Andrew Arnott, Dirk Balfanz, Aiden Bell, John Bradley, Marcos Caceres, Brian Campbell, Scott Cantor, Blaine Cook, Roger Crew, Leah Culver, Bill de hOra, Andre DeMarre, Brian Eaton, Wesley Eddy, Wolter Eldering, Brian Ellin, Igor Faynberg, George Fletcher, Tim Freeman, Luca Frosini, Evan Gilbert, Yaron Y. Goland, Brent Goldman, Kristoffer Gronowski, Eran Hammer, Dick Hardt, Justin Hart, Craig Heath, Phil Hunt, Michael B. Jones, Terry Jones, John Kemp, Mark Kent, Raffi Krikorian, Chasen Le Hara, Rasmus Lerdorf, Torsten Lodderstedt, Hui-Lan Lu, Casey Lucas, Paul Madsen, Alastair Mair, Eve Maler, James Manger, Mark McGloin, Laurence Miao, William Mills, Chuck Mortimore, Anthony Nadalin, Julian Reschke, Justin Richer, Peter Saint-Andre, Nat Sakimura, Rob Sayre, Marius Scurtescu, Naitik Shah, Luke Shepard, Vlad Skvortsov, Justin Smith, Haibin Song, Niv Steingarten, Christian Stuebner, Jeremy Suriel, Paul Tarjan, Christopher Thomas, Henry S. Thompson, Allen Tom, Franklin Tse, Nick Walker, Shane Weeden, and Skylar Woodward.

This document was produced under the chairmanship of Blaine Cook, Peter Saint-Andre, Hannes Tschofenig, Barry Leiba, and Derek Atkins. The area directors included Lisa Dusseault, Peter Saint-Andre, and Stephen Farrell.

Author's Address

Dick Hardt (editor)
Microsoft

EMail: dick.hardt@gmail.com
URI: http://dickhardt.org/

Chapter 2

RFC 6750: OAuth 2.0 Bearer Tokens

Access token usage is defined in RFC 6750, although the format of access tokens isn't defined here. This spec defines "Bearer Tokens", which just means that it's a type of token that can be used by whoever has the token with no additional information. The particular format access tokens take (random strings, JWTs, etc) is not relevant to OAuth clients so isn't included in this spec. Only the Authorization Server and Resource Server need to coordinate on access token formats, so that is left up to the particular implementation or a future spec.

Bearer tokens were a major point of contention in the early days of OAuth. Some people wanted the simplicity of bearer tokens, others wanted tokens that required some sort of cryptographic binding with the client. The result was that access tokens were taken out of the core spec entirely rather than coming to a compromise.

While today most deployed systems use bearer tokens, there is still a need for some sort of cryptographic token type that would prevent attackers from being able to use access tokens if stolen. There are several proposed new ways to accomplish this, although none are formally recognized as a standard, and this is a space where continuing work is happening within the OAuth group.

Internet Engineering Task Force (IETF)
Request for Comments: 6750
Category: Standards Track
ISSN: 2070-1721

M. Jones
Microsoft
D. Hardt
Independent

October 2012

The OAuth 2.0 Authorization Framework: Bearer Token Usage

Abstract

This specification describes how to use bearer tokens in HTTP requests to access OAuth 2.0 protected resources. Any party in possession of a bearer token (a "bearer") can use it to get access to the associated resources (without demonstrating possession of a cryptographic key). To prevent misuse, bearer tokens need to be protected from disclosure in storage and in transport.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc6750.

Copyright Notice

Copyright (c) 2012 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. Notational Conventions 3
   1.2. Terminology 3
   1.3. Overview 3
2. Authenticated Requests 4
   2.1. Authorization Request Header Field 5
   2.2. Form-Encoded Body Parameter 5
   2.3. URI Query Parameter 6
3. The WWW-Authenticate Response Header Field 7
   3.1. Error Codes 9
4. Example Access Token Response 10
5. Security Considerations 10
   5.1. Security Threats 10
   5.2. Threat Mitigation 11
   5.3. Summary of Recommendations 13
6. IANA Considerations 14
   6.1. OAuth Access Token Type Registration 14
        6.1.1. The "Bearer" OAuth Access Token Type 14
   6.2. OAuth Extensions Error Registration 14
        6.2.1. The "invalid_request" Error Value 14
        6.2.2. The "invalid_token" Error Value 15
        6.2.3. The "insufficient_scope" Error Value 15
7. References 15
   7.1. Normative References 15
   7.2. Informative References 17
Appendix A. Acknowledgements 18

1. Introduction

OAuth enables clients to access protected resources by obtaining an access token, which is defined in "The OAuth 2.0 Authorization Framework" [RFC6749] as "a string representing an access authorization issued to the client", rather than using the resource owner's credentials directly.

Tokens are issued to clients by an authorization server with the approval of the resource owner. The client uses the access token to access the protected resources hosted by the resource server. This specification describes how to make protected resource requests when the OAuth access token is a bearer token.

This specification defines the use of bearer tokens over HTTP/1.1 [RFC2616] using Transport Layer Security (TLS) [RFC5246] to access protected resources. TLS is mandatory to implement and use with this specification; other specifications may extend this specification for use with other protocols. While designed for use with access tokens

resulting from OAuth 2.0 authorization [RFC6749] flows to access OAuth protected resources, this specification actually defines a general HTTP authorization method that can be used with bearer tokens from any source to access any resources protected by those bearer tokens. The Bearer authentication scheme is intended primarily for server authentication using the WWW-Authenticate and Authorization HTTP headers but does not preclude its use for proxy authentication.

1.1. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD 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].

This document uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234]. Additionally, the following rules are included from HTTP/1.1 [RFC2617]: auth-param and auth-scheme; and from "Uniform Resource Identifier (URI): Generic Syntax" [RFC3986]: URI-reference.

Unless otherwise noted, all the protocol parameter names and values are case sensitive.

1.2. Terminology

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

All other terms are as defined in "The OAuth 2.0 Authorization Framework" [RFC6749].

1.3. Overview

OAuth provides a method for clients to access a protected resource on behalf of a resource owner. In the general case, before a client can access a protected resource, it must first obtain an authorization grant from the resource owner and then exchange the authorization grant for an access token. The access token represents the grant's scope, duration, and other attributes granted by the authorization grant. The client accesses the protected resource by presenting the access token to the resource server. In some cases, a client can directly present its own credentials to an authorization server to obtain an access token without having to first obtain an authorization grant from a resource owner.

The access token provides an abstraction, replacing different authorization constructs (e.g., username and password, assertion) for a single token understood by the resource server. This abstraction enables issuing access tokens valid for a short time period, as well as removing the resource server's need to understand a wide range of authentication schemes.

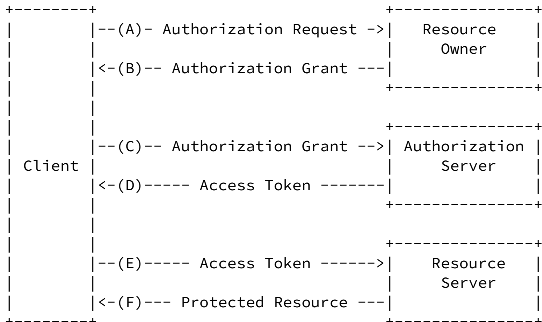


Figure 1: Abstract Protocol Flow

The abstract OAuth 2.0 flow illustrated in Figure 1 describes the interaction between the client, resource owner, authorization server, and resource server (described in [RFC6749]). The following two steps are specified within this document:

(E) The client requests the protected resource from the resource server and authenticates by presenting the access token.

(F) The resource server validates the access token, and if valid, serves the request.

This document also imposes semantic requirements upon the access token returned in step (D).

2. Authenticated Requests

This section defines three methods of sending bearer access tokens in resource requests to resource servers. Clients MUST NOT use more than one method to transmit the token in each request.

2.1. Authorization Request Header Field

When sending the access token in the "Authorization" request header field defined by HTTP/1.1 [RFC2617], the client uses the "Bearer" authentication scheme to transmit the access token.

For example:

```
GET /resource HTTP/1.1
Host: server.example.com
Authorization: Bearer mF_9.B5f-4.1JqM
```

The syntax of the "Authorization" header field for this scheme follows the usage of the Basic scheme defined in Section 2 of [RFC2617]. Note that, as with Basic, it does not conform to the generic syntax defined in Section 1.2 of [RFC2617] but is compatible with the general authentication framework being developed for HTTP 1.1 [HTTP-AUTH], although it does not follow the preferred practice outlined therein in order to reflect existing deployments. The syntax for Bearer credentials is as follows:

```
b64token    = 1*( ALPHA / DIGIT /
                  "-" / "." / "_" / "~" / "+" / "/" ) *"="
credentials = "Bearer" 1*SP b64token
```

Clients SHOULD make authenticated requests with a bearer token using the "Authorization" request header field with the "Bearer" HTTP authorization scheme. Resource servers MUST support this method.

2.2. Form-Encoded Body Parameter

When sending the access token in the HTTP request entity-body, the client adds the access token to the request-body using the "access_token" parameter. The client MUST NOT use this method unless all of the following conditions are met:

- o The HTTP request entity-header includes the "Content-Type" header field set to "application/x-www-form-urlencoded".
- o The entity-body follows the encoding requirements of the "application/x-www-form-urlencoded" content-type as defined by HTML 4.01 [W3C.REC-html401-19991224].
- o The HTTP request entity-body is single-part.

- o The content to be encoded in the entity-body MUST consist entirely of ASCII [USASCII] characters.

- o The HTTP request method is one for which the request-body has defined semantics. In particular, this means that the "GET" method MUST NOT be used.

The entity-body MAY include other request-specific parameters, in which case the "access_token" parameter MUST be properly separated from the request-specific parameters using "&" character(s) (ASCII code 38).

For example, the client makes the following HTTP request using transport-layer security:

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

access_token=mF_9.B5f-4.1JqM
```

The "application/x-www-form-urlencoded" method SHOULD NOT be used except in application contexts where participating browsers do not have access to the "Authorization" request header field. Resource servers MAY support this method.

2.3. URI Query Parameter

When sending the access token in the HTTP request URI, the client adds the access token to the request URI query component as defined by "Uniform Resource Identifier (URI): Generic Syntax" [RFC3986], using the "access_token" parameter.

For example, the client makes the following HTTP request using transport-layer security:

```
GET /resource?access_token=mF_9.B5f-4.1JqM HTTP/1.1
Host: server.example.com
```

The HTTP request URI query can include other request-specific parameters, in which case the "access_token" parameter MUST be properly separated from the request-specific parameters using "&" character(s) (ASCII code 38).

For example:

```
https://server.example.com/resource?access_token=mF_9.B5f-4.1JqM&p=q
```

Clients using the URI Query Parameter method SHOULD also send a Cache-Control header containing the "no-store" option. Server success (2XX status) responses to these requests SHOULD contain a Cache-Control header with the "private" option.

Because of the security weaknesses associated with the URI method (see Section 5), including the high likelihood that the URL containing the access token will be logged, it SHOULD NOT be used unless it is impossible to transport the access token in the "Authorization" request header field or the HTTP request entity-body. Resource servers MAY support this method.

This method is included to document current use; its use is not recommended, due to its security deficiencies (see Section 5) and also because it uses a reserved query parameter name, which is counter to URI namespace best practices, per "Architecture of the World Wide Web, Volume One" [W3C.REC-webarch-20041215].

3. The WWW-Authenticate Response Header Field

If the protected resource request does not include authentication credentials or does not contain an access token that enables access to the protected resource, the resource server MUST include the HTTP "WWW-Authenticate" response header field; it MAY include it in response to other conditions as well. The "WWW-Authenticate" header field uses the framework defined by HTTP/1.1 [RFC2617].

All challenges defined by this specification MUST use the auth-scheme value "Bearer". This scheme MUST be followed by one or more auth-param values. The auth-param attributes used or defined by this specification are as follows. Other auth-param attributes MAY be used as well.

A "realm" attribute MAY be included to indicate the scope of protection in the manner described in HTTP/1.1 [RFC2617]. The "realm" attribute MUST NOT appear more than once.

The "scope" attribute is defined in Section 3.3 of [RFC6749]. The "scope" attribute is a space-delimited list of case-sensitive scope values indicating the required scope of the access token for accessing the requested resource. "scope" values are implementation defined; there is no centralized registry for them; allowed values are defined by the authorization server. The order of "scope" values is not significant. In some cases, the "scope" value will be used

when requesting a new access token with sufficient scope of access to utilize the protected resource. Use of the "scope" attribute is OPTIONAL. The "scope" attribute MUST NOT appear more than once. The "scope" value is intended for programmatic use and is not meant to be displayed to end-users.

Two example scope values follow; these are taken from the OpenID Connect [OpenID.Messages] and the Open Authentication Technology Committee (OATC) Online Multimedia Authorization Protocol [OMAP] OAuth 2.0 use cases, respectively:

```
scope="openid profile email"
scope="urn:example:channel=HBO&urn:example:rating=G,PG-13"
```

If the protected resource request included an access token and failed authentication, the resource server SHOULD include the "error" attribute to provide the client with the reason why the access request was declined. The parameter value is described in Section 3.1. In addition, the resource server MAY include the "error_description" attribute to provide developers a human-readable explanation that is not meant to be displayed to end-users. It also MAY include the "error_uri" attribute with an absolute URI identifying a human-readable web page explaining the error. The "error", "error_description", and "error_uri" attributes MUST NOT appear more than once.

Values for the "scope" attribute (specified in Appendix A.4 of [RFC6749]) MUST NOT include characters outside the set %x21 / %x23-5B / %x5D-7E for representing scope values and %x20 for delimiters between scope values. Values for the "error" and "error_description" attributes (specified in Appendixes A.7 and A.8 of [RFC6749]) MUST NOT include characters outside the set %x20-21 / %x23-5B / %x5D-7E. Values for the "error_uri" attribute (specified in Appendix A.9 of [RFC6749]) MUST conform to the URI-reference syntax and thus MUST NOT include characters outside the set %x21 / %x23-5B / %x5D-7E.

For example, in response to a protected resource request without authentication:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="example"
```

And in response to a protected resource request with an authentication attempt using an expired access token:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="example",
                  error="invalid_token",
                  error_description="The access token expired"
```

3.1. Error Codes

When a request fails, the resource server responds using the appropriate HTTP status code (typically, 400, 401, 403, or 405) and includes one of the following error codes in the response:

invalid_request
: The request is missing a required parameter, includes an unsupported parameter or parameter value, repeats the same parameter, uses more than one method for including an access token, or is otherwise malformed. The resource server SHOULD respond with the HTTP 400 (Bad Request) status code.

invalid_token
: The access token provided is expired, revoked, malformed, or invalid for other reasons. The resource SHOULD respond with the HTTP 401 (Unauthorized) status code. The client MAY request a new access token and retry the protected resource request.

insufficient_scope
: The request requires higher privileges than provided by the access token. The resource server SHOULD respond with the HTTP 403 (Forbidden) status code and MAY include the "scope" attribute with the scope necessary to access the protected resource.

If the request lacks any authentication information (e.g., the client was unaware that authentication is necessary or attempted using an unsupported authentication method), the resource server SHOULD NOT include an error code or other error information.

For example:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="example"
```

4. Example Access Token Response

Typically, a bearer token is returned to the client as part of an OAuth 2.0 [RFC6749] access token response. An example of such a response is:

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "access_token":"mF_9.B5f-4.1JqM",
  "token_type":"Bearer",
  "expires_in":3600,
  "refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA"
}
```

5. Security Considerations

This section describes the relevant security threats regarding token handling when using bearer tokens and describes how to mitigate these threats.

5.1. Security Threats

The following list presents several common threats against protocols utilizing some form of tokens. This list of threats is based on NIST Special Publication 800-63 [NIST800-63]. Since this document builds on the OAuth 2.0 Authorization specification [RFC6749], we exclude a discussion of threats that are described there or in related documents.

Token manufacture/modification: An attacker may generate a bogus token or modify the token contents (such as the authentication or attribute statements) of an existing token, causing the resource server to grant inappropriate access to the client. For example, an attacker may modify the token to extend the validity period; a malicious client may modify the assertion to gain access to information that they should not be able to view.

Token disclosure: Tokens may contain authentication and attribute statements that include sensitive information.

Token redirect: An attacker uses a token generated for consumption by one resource server to gain access to a different resource server that mistakenly believes the token to be for it.

Token replay: An attacker attempts to use a token that has already been used with that resource server in the past.

5.2. Threat Mitigation

A large range of threats can be mitigated by protecting the contents of the token by using a digital signature or a Message Authentication Code (MAC). Alternatively, a bearer token can contain a reference to authorization information, rather than encoding the information directly. Such references MUST be infeasible for an attacker to guess; using a reference may require an extra interaction between a server and the token issuer to resolve the reference to the authorization information. The mechanics of such an interaction are not defined by this specification.

This document does not specify the encoding or the contents of the token; hence, detailed recommendations about the means of guaranteeing token integrity protection are outside the scope of this document. The token integrity protection MUST be sufficient to prevent the token from being modified.

To deal with token redirect, it is important for the authorization server to include the identity of the intended recipients (the audience), typically a single resource server (or a list of resource servers), in the token. Restricting the use of the token to a specific scope is also RECOMMENDED.

The authorization server MUST implement TLS. Which version(s) ought to be implemented will vary over time and will depend on the widespread deployment and known security vulnerabilities at the time of implementation. At the time of this writing, TLS version 1.2 [RFC5246] is the most recent version, but it has very limited actual deployment and might not be readily available in implementation toolkits. TLS version 1.0 [RFC2246] is the most widely deployed version and will give the broadest interoperability.

To protect against token disclosure, confidentiality protection MUST be applied using TLS [RFC5246] with a ciphersuite that provides confidentiality and integrity protection. This requires that the communication interaction between the client and the authorization server, as well as the interaction between the client and the resource server, utilize confidentiality and integrity protection. Since TLS is mandatory to implement and to use with this specification, it is the preferred approach for preventing token

disclosure via the communication channel. For those cases where the client is prevented from observing the contents of the token, token encryption MUST be applied in addition to the usage of TLS protection. As a further defense against token disclosure, the client MUST validate the TLS certificate chain when making requests to protected resources, including checking the Certificate Revocation List (CRL) [RFC5280].

Cookies are typically transmitted in the clear. Thus, any information contained in them is at risk of disclosure. Therefore, bearer tokens MUST NOT be stored in cookies that can be sent in the clear. See "HTTP State Management Mechanism" [RFC6265] for security considerations about cookies.

In some deployments, including those utilizing load balancers, the TLS connection to the resource server terminates prior to the actual server that provides the resource. This could leave the token unprotected between the front-end server where the TLS connection terminates and the back-end server that provides the resource. In such deployments, sufficient measures MUST be employed to ensure confidentiality of the token between the front-end and back-end servers; encryption of the token is one such possible measure.

To deal with token capture and replay, the following recommendations are made: First, the lifetime of the token MUST be limited; one means of achieving this is by putting a validity time field inside the protected part of the token. Note that using short-lived (one hour or less) tokens reduces the impact of them being leaked. Second, confidentiality protection of the exchanges between the client and the authorization server and between the client and the resource server MUST be applied. As a consequence, no eavesdropper along the communication path is able to observe the token exchange. Consequently, such an on-path adversary cannot replay the token. Furthermore, when presenting the token to a resource server, the client MUST verify the identity of that resource server, as per Section 3.1 of "HTTP Over TLS" [RFC2818]. Note that the client MUST validate the TLS certificate chain when making these requests to protected resources. Presenting the token to an unauthenticated and unauthorized resource server or failing to validate the certificate chain will allow adversaries to steal the token and gain unauthorized access to protected resources.

5.3. Summary of Recommendations

Safeguard bearer tokens: Client implementations MUST ensure that bearer tokens are not leaked to unintended parties, as they will be able to use them to gain access to protected resources. This is the primary security consideration when using bearer tokens and underlies all the more specific recommendations that follow.

Validate TLS certificate chains: The client MUST validate the TLS certificate chain when making requests to protected resources. Failing to do so may enable DNS hijacking attacks to steal the token and gain unintended access.

Always use TLS (https): Clients MUST always use TLS [RFC5246] (https) or equivalent transport security when making requests with bearer tokens. Failing to do so exposes the token to numerous attacks that could give attackers unintended access.

Don't store bearer tokens in cookies: Implementations MUST NOT store bearer tokens within cookies that can be sent in the clear (which is the default transmission mode for cookies). Implementations that do store bearer tokens in cookies MUST take precautions against cross-site request forgery.

Issue short-lived bearer tokens: Token servers SHOULD issue short-lived (one hour or less) bearer tokens, particularly when issuing tokens to clients that run within a web browser or other environments where information leakage may occur. Using short-lived bearer tokens can reduce the impact of them being leaked.

Issue scoped bearer tokens: Token servers SHOULD issue bearer tokens that contain an audience restriction, scoping their use to the intended relying party or set of relying parties.

Don't pass bearer tokens in page URLs: Bearer tokens SHOULD NOT be passed in page URLs (for example, as query string parameters). Instead, bearer tokens SHOULD be passed in HTTP message headers or message bodies for which confidentiality measures are taken. Browsers, web servers, and other software may not adequately secure URLs in the browser history, web server logs, and other data structures. If bearer tokens are passed in page URLs, attackers might be able to steal them from the history data, logs, or other unsecured locations.

6. IANA Considerations

6.1. OAuth Access Token Type Registration

This specification registers the following access token type in the OAuth Access Token Types registry defined in [RFC6749].

6.1.1. The "Bearer" OAuth Access Token Type

Type name:
  Bearer

Additional Token Endpoint Response Parameters:
  (none)

HTTP Authentication Scheme(s):
  Bearer

Change controller:
  IETF

Specification document(s):
  RFC 6750

6.2. OAuth Extensions Error Registration

This specification registers the following error values in the OAuth Extensions Error registry defined in [RFC6749].

6.2.1. The "invalid_request" Error Value

Error name:
  invalid_request

Error usage location:
  Resource access error response

Related protocol extension:
  Bearer access token type

Change controller:
  IETF

Specification document(s):
  RFC 6750

6.2.2. The "invalid_token" Error Value

Error name:
invalid_token

Error usage location:
Resource access error response

Related protocol extension:
Bearer access token type

Change controller:
IETF

Specification document(s):
RFC 6750

6.2.3. The "insufficient_scope" Error Value

Error name:
insufficient_scope

Error usage location:
Resource access error response

Related protocol extension:
Bearer access token type

Change controller:
IETF

Specification document(s):
RFC 6750

7. References

7.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2246] Dierks, T. and C. Allen, "The TLS Protocol Version 1.0", RFC 2246, January 1999.

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[RFC2617] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and L. Stewart, "HTTP Authentication: Basic and Digest Access Authentication", RFC 2617, June 1999.

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.

[RFC5234] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

[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, May 2008.

[RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, April 2011.

[RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, October 2012.

[USASCII] American National Standards Institute, "Coded Character Set -- 7-bit American Standard Code for Information Interchange", ANSI X3.4, 1986.

[W3C.REC-html401-19991224]
Raggett, D., Le Hors, A., and I. Jacobs, "HTML 4.01 Specification", World Wide Web Consortium Recommendation REC-html401-19991224, December 1999,
.

[W3C.REC-webarch-20041215]
Jacobs, I. and N. Walsh, "Architecture of the World Wide Web, Volume One", World Wide Web Consortium Recommendation REC-webarch-20041215, December 2004,
.

7.2. Informative References

[HTTP-AUTH] Fielding, R., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Authentication", Work in Progress, October 2012.

[NIST800-63] Burr, W., Dodson, D., Newton, E., Perlner, R., Polk, T., Gupta, S., and E. Nabbus, "NIST Special Publication 800-63-1, INFORMATION SECURITY", December 2011, .

[OMAP] Huff, J., Schlacht, D., Nadalin, A., Simmons, J., Rosenberg, P., Madsen, P., Ace, T., Rickelton-Abdi, C., and B. Boyer, "Online Multimedia Authorization Protocol: An Industry Standard for Authorized Access to Internet Multimedia Resources", April 2012, .

[OpenID.Messages] Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., Mortimore, C., and E. Jay, "OpenID Connect Messages 1.0", June 2012, .

Appendix A. Acknowledgements

The following people contributed to preliminary versions of this document: Blaine Cook (BT), Brian Eaton (Google), Yaron Y. Goland (Microsoft), Brent Goldman (Facebook), Raffi Krikorian (Twitter), Luke Shepard (Facebook), and Allen Tom (Yahoo!). The content and concepts within are a product of the OAuth community, the Web Resource Authorization Profiles (WRAP) community, and the OAuth Working Group. David Recordon created a preliminary version of this specification based upon an early draft of the specification that evolved into OAuth 2.0 [RFC6749]. Michael B. Jones in turn created the first version (00) of this specification using portions of David's preliminary document and edited all subsequent versions.

The OAuth Working Group has dozens of very active contributors who proposed ideas and wording for this document, including Michael Adams, Amanda Anganes, Andrew Arnott, Derek Atkins, Dirk Balfanz, John Bradley, Brian Campbell, Francisco Corella, Leah Culver, Bill de hOra, Breno de Medeiros, Brian Ellin, Stephen Farrell, Igor Faynberg, George Fletcher, Tim Freeman, Evan Gilbert, Yaron Y. Goland, Eran Hammer, Thomas Hardjono, Dick Hardt, Justin Hart, Phil Hunt, John Kemp, Chasen Le Hara, Barry Leiba, Amos Jeffries, Michael B. Jones, Torsten Lodderstedt, Paul Madsen, Eve Maler, James Manger, Laurence Miao, William J. Mills, Chuck Mortimore, Anthony Nadalin, Axel Nennker, Mark Nottingham, David Recordon, Julian Reschke, Rob Richards, Justin Richer, Peter Saint-Andre, Nat Sakimura, Rob Sayre, Marius Scurtescu, Naitik Shah, Justin Smith, Christian Stuebner, Jeremy Suriel, Doug Tangren, Paul Tarjan, Hannes Tschofenig, Franklin Tse, Sean Turner, Paul Walker, Shane Weeden, Skylar Woodward, and Zachary Zeltsan.

Authors' Addresses

Michael B. Jones
Microsoft

EMail: mbj@microsoft.com
URI: http://self-issued.info/

Dick Hardt
Independent

EMail: dick.hardt@gmail.com
URI: http://dickhardt.org/

Chapter 3

RFC 7636: Proof Key for Code Exchange (PKCE)

The Authorization Code flow traditionally relies on a pre-established client secret for its security. Platforms such as mobile apps and JavaScript apps have no ability to deploy a client secret, so they would previously have been unable to implement a secure Authorization Code flow.

PKCE is an extension to the Authorization Code flow that adds a secure link between starting and completing the flow so that clients can use it without a preconfigured secret.

PKCE works by the app first generating a new secret each time it starts the Authorization Code flow, and it sends a hash of the secret in the initial authorization request. The original secret is then required in order to exchange the authorization code for an access token, ensuring that even if an attacker can steal the authorization code, they would be unable to use it.

At the time of publication, PKCE was recommended for mobile apps, but it has proven to be useful even for JavaScript apps, and now the latest Security Best Current Practice recommends using it for all types of apps, even apps with a client secret.

Internet Engineering Task Force (IETF)
Request for Comments: 7636
Category: Standards Track
ISSN: 2070-1721

N. Sakimura, Ed.
Nomura Research Institute
J. Bradley
Ping Identity
N. Agarwal
Google

September 2015

Proof Key for Code Exchange by OAuth Public Clients

Abstract

OAuth 2.0 public clients utilizing the Authorization Code Grant are susceptible to the authorization code interception attack. This specification describes the attack as well as a technique to mitigate against the threat through the use of Proof Key for Code Exchange (PKCE, pronounced "pixy").

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc7636.

Copyright Notice

Copyright (c) 2015 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. Protocol Flow 5
2. Notational Conventions 6
3. Terminology 7
   3.1. Abbreviations 7
4. Protocol 8
   4.1. Client Creates a Code Verifier 8
   4.2. Client Creates the Code Challenge 8
   4.3. Client Sends the Code Challenge with the Authorization Request 9
   4.4. Server Returns the Code 9
      4.4.1. Error Response 9
   4.5. Client Sends the Authorization Code and the Code Verifier to the Token Endpoint 10
   4.6. Server Verifies code_verifier before Returning the Tokens 10
5. Compatibility 11
6. IANA Considerations 11
   6.1. OAuth Parameters Registry 11
   6.2. PKCE Code Challenge Method Registry 11
      6.2.1. Registration Template 12
      6.2.2. Initial Registry Contents 13
7. Security Considerations 13
   7.1. Entropy of the code_verifier 13
   7.2. Protection against Eavesdroppers 13
   7.3. Salting the code_challenge 14
   7.4. OAuth Security Considerations 14
   7.5. TLS Security Considerations 15
8. References 15
   8.1. Normative References 15
   8.2. Informative References 16
Appendix A. Notes on Implementing Base64url Encoding without Padding 17
Appendix B. Example for the S256 code_challenge_method 17
Acknowledgements 19
Authors' Addresses 20

1. Introduction

OAuth 2.0 [RFC6749] public clients are susceptible to the authorization code interception attack.

In this attack, the attacker intercepts the authorization code returned from the authorization endpoint within a communication path not protected by Transport Layer Security (TLS), such as inter-application communication within the client's operating system.

Once the attacker has gained access to the authorization code, it can use it to obtain the access token.

Figure 1 shows the attack graphically. In step (1), the native application running on the end device, such as a smartphone, issues an OAuth 2.0 Authorization Request via the browser/operating system. The Redirection Endpoint URI in this case typically uses a custom URI scheme. Step (1) happens through a secure API that cannot be intercepted, though it may potentially be observed in advanced attack scenarios. The request then gets forwarded to the OAuth 2.0 authorization server in step (2). Because OAuth requires the use of TLS, this communication is protected by TLS and cannot be intercepted. The authorization server returns the authorization code in step (3). In step (4), the Authorization Code is returned to the requester via the Redirection Endpoint URI that was provided in step (1).

Note that it is possible for a malicious app to register itself as a handler for the custom scheme in addition to the legitimate OAuth 2.0 app. Once it does so, the malicious app is now able to intercept the authorization code in step (4). This allows the attacker to request and obtain an access token in steps (5) and (6), respectively.

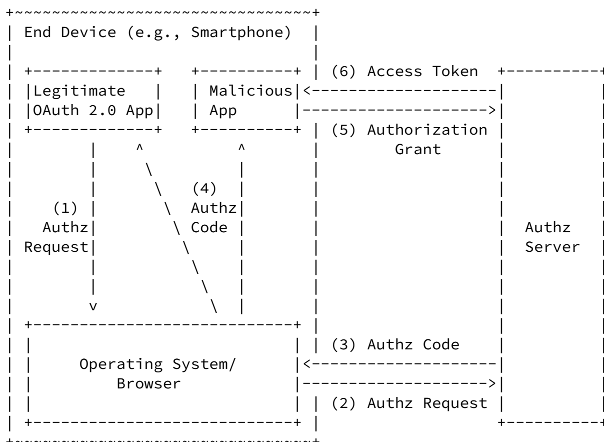


Figure 1: Authorization Code Interception Attack

A number of pre-conditions need to hold for this attack to work:

1. The attacker manages to register a malicious application on the client device and registers a custom URI scheme that is also used by another application. The operating systems must allow a custom URI scheme to be registered by multiple applications.

2. The OAuth 2.0 authorization code grant is used.

3. The attacker has access to the OAuth 2.0 [RFC6749] "client_id" and "client_secret" (if provisioned). All OAuth 2.0 native app client-instances use the same "client_id". Secrets provisioned in client binary applications cannot be considered confidential.

4. Either one of the following condition is met:

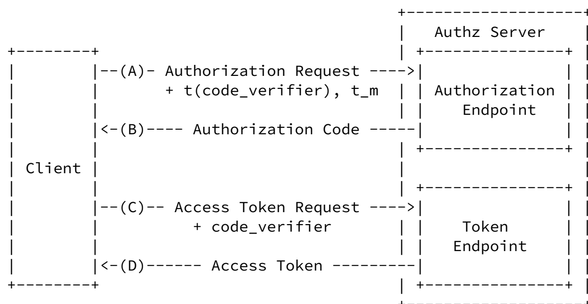
   4a. The attacker (via the installed application) is able to observe only the responses from the authorization endpoint. When "code_challenge_method" value is "plain", only this attack is mitigated.

4b. A more sophisticated attack scenario allows the attacker to observe requests (in addition to responses) to the authorization endpoint. The attacker is, however, not able to act as a man in the middle. This was caused by leaking http log information in the OS. To mitigate this, "code_challenge_method" value must be set either to "S256" or a value defined by a cryptographically secure "code_challenge_method" extension.

While this is a long list of pre-conditions, the described attack has been observed in the wild and has to be considered in OAuth 2.0 deployments. While the OAuth 2.0 threat model (Section 4.4.1 of [RFC6819]) describes mitigation techniques, they are, unfortunately, not applicable since they rely on a per-client instance secret or a per-client instance redirect URI.

To mitigate this attack, this extension utilizes a dynamically created cryptographically random key called "code verifier". A unique code verifier is created for every authorization request, and its transformed value, called "code challenge", is sent to the authorization server to obtain the authorization code. The authorization code obtained is then sent to the token endpoint with the "code verifier", and the server compares it with the previously received request code so that it can perform the proof of possession of the "code verifier" by the client. This works as the mitigation since the attacker would not know this one-time key, since it is sent over TLS and cannot be intercepted.

1.1. Protocol Flow



```
                                                 +-------------------+
                                                 |   Authz Server    |
       +--------+                                | +---------------+ |
       |        |--(A)- Authorization Request ---->|               | |
       |        |       + t(code_verifier), t_m  | | Authorization | |
       |        |                                | |    Endpoint   | |
       |        |<-(B)---- Authorization Code -----|               | |
       |        |                                | +---------------+ |
       | Client |                                |                   |
       |        |                                | +---------------+ |
       |        |--(C)-- Access Token Request ---->|               | |
       |        |          + code_verifier       | |    Token      | |
       |        |                                | |   Endpoint    | |
       |        |<-(D)------ Access Token ---------|               | |
       +--------+                                | +---------------+ |
                                                 +-------------------+
```

Figure 2: Abstract Protocol Flow

This specification adds additional parameters to the OAuth 2.0 Authorization and Access Token Requests, shown in abstract form in Figure 2.

A. The client creates and records a secret named the "code_verifier" and derives a transformed version "t(code_verifier)" (referred to as the "code_challenge"), which is sent in the OAuth 2.0 Authorization Request along with the transformation method "t_m".

B. The Authorization Endpoint responds as usual but records "t(code_verifier)" and the transformation method.

C. The client then sends the authorization code in the Access Token Request as usual but includes the "code_verifier" secret generated at (A).

D. The authorization server transforms "code_verifier" and compares it to "t(code_verifier)" from (B). Access is denied if they are not equal.

An attacker who intercepts the authorization code at (B) is unable to redeem it for an access token, as they are not in possession of the "code_verifier" secret.

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 natural language meanings.

This specification uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234].

STRING denotes a sequence of zero or more ASCII [RFC20] characters.

OCTETS denotes a sequence of zero or more octets.

ASCII(STRING) denotes the octets of the ASCII [RFC20] representation of STRING where STRING is a sequence of zero or more ASCII characters.

BASE64URL-ENCODE(OCTETS) denotes the base64url encoding of OCTETS, per Appendix A, producing a STRING.

BASE64URL-DECODE(STRING) denotes the base64url decoding of STRING, per Appendix A, producing a sequence of octets.

SHA256(OCTETS) denotes a SHA2 256-bit hash [RFC6234] of OCTETS.

3. Terminology

In addition to the terms defined in OAuth 2.0 [RFC6749], this specification defines the following terms:

code verifier
: A cryptographically random string that is used to correlate the authorization request to the token request.

code challenge
: A challenge derived from the code verifier that is sent in the authorization request, to be verified against later.

code challenge method
: A method that was used to derive code challenge.

Base64url Encoding
: Base64 encoding using the URL- and filename-safe character set defined in Section 5 of [RFC4648], with all trailing '=' characters omitted (as permitted by Section 3.2 of [RFC4648]) and without the inclusion of any line breaks, whitespace, or other additional characters. (See Appendix A for notes on implementing base64url encoding without padding.)

3.1. Abbreviations

ABNF Augmented Backus-Naur Form

Authz Authorization

PKCE Proof Key for Code Exchange

MITM Man-in-the-middle

MTI Mandatory To Implement

4. Protocol

4.1. Client Creates a Code Verifier

The client first creates a code verifier, "code_verifier", for each OAuth 2.0 [RFC6749] Authorization Request, in the following manner:

code_verifier = high-entropy cryptographic random STRING using the unreserved characters [A-Z] / [a-z] / [0-9] / "-" / "." / "_" / "~" from Section 2.3 of [RFC3986], with a minimum length of 43 characters and a maximum length of 128 characters.

ABNF for "code_verifier" is as follows.

```
code-verifier = 43*128unreserved
unreserved = ALPHA / DIGIT / "-" / "." / "_" / "~"
ALPHA = %x41-5A / %x61-7A
DIGIT = %x30-39
```

NOTE: The code verifier SHOULD have enough entropy to make it impractical to guess the value. It is RECOMMENDED that the output of a suitable random number generator be used to create a 32-octet sequence. The octet sequence is then base64url-encoded to produce a 43-octet URL safe string to use as the code verifier.

4.2. Client Creates the Code Challenge

The client then creates a code challenge derived from the code verifier by using one of the following transformations on the code verifier:

plain
   code_challenge = code_verifier

S256
   code_challenge = BASE64URL-ENCODE(SHA256(ASCII(code_verifier)))

If the client is capable of using "S256", it MUST use "S256", as "S256" is Mandatory To Implement (MTI) on the server. Clients are permitted to use "plain" only if they cannot support "S256" for some technical reason and know via out-of-band configuration that the server supports "plain".

The plain transformation is for compatibility with existing deployments and for constrained environments that can't use the S256 transformation.

ABNF for "code_challenge" is as follows.

```
code-challenge = 43*128unreserved
unreserved = ALPHA / DIGIT / "-" / "." / "_" / "~"
ALPHA = %x41-5A / %x61-7A
DIGIT = %x30-39
```

4.3. Client Sends the Code Challenge with the Authorization Request

The client sends the code challenge as part of the OAuth 2.0 Authorization Request (Section 4.1.1 of [RFC6749]) using the following additional parameters:

code_challenge
REQUIRED. Code challenge.

code_challenge_method
OPTIONAL, defaults to "plain" if not present in the request. Code verifier transformation method is "S256" or "plain".

4.4. Server Returns the Code

When the server issues the authorization code in the authorization response, it MUST associate the "code_challenge" and "code_challenge_method" values with the authorization code so it can be verified later.

Typically, the "code_challenge" and "code_challenge_method" values are stored in encrypted form in the "code" itself but could alternatively be stored on the server associated with the code. The server MUST NOT include the "code_challenge" value in client requests in a form that other entities can extract.

The exact method that the server uses to associate the "code_challenge" with the issued "code" is out of scope for this specification.

4.4.1. Error Response

If the server requires Proof Key for Code Exchange (PKCE) by OAuth public clients and the client does not send the "code_challenge" in the request, the authorization endpoint MUST return the authorization error response with the "error" value set to "invalid_request". The "error_description" or the response of "error_uri" SHOULD explain the nature of error, e.g., code challenge required.

If the server supporting PKCE does not support the requested transformation, the authorization endpoint MUST return the authorization error response with "error" value set to "invalid_request". The "error_description" or the response of "error_uri" SHOULD explain the nature of error, e.g., transform algorithm not supported.

4.5. Client Sends the Authorization Code and the Code Verifier to the Token Endpoint

Upon receipt of the Authorization Code, the client sends the Access Token Request to the token endpoint. In addition to the parameters defined in the OAuth 2.0 Access Token Request (Section 4.1.3 of [RFC6749]), it sends the following parameter:

code_verifier
 REQUIRED. Code verifier

The "code_challenge_method" is bound to the Authorization Code when the Authorization Code is issued. That is the method that the token endpoint MUST use to verify the "code_verifier".

4.6. Server Verifies code_verifier before Returning the Tokens

Upon receipt of the request at the token endpoint, the server verifies it by calculating the code challenge from the received "code_verifier" and comparing it with the previously associated "code_challenge", after first transforming it according to the "code_challenge_method" method specified by the client.

If the "code_challenge_method" from Section 4.3 was "S256", the received "code_verifier" is hashed by SHA-256, base64url-encoded, and then compared to the "code_challenge", i.e.:

```
BASE64URL-ENCODE(SHA256(ASCII(code_verifier))) == code_challenge
```

If the "code_challenge_method" from Section 4.3 was "plain", they are compared directly, i.e.:

```
code_verifier == code_challenge.
```

If the values are equal, the token endpoint MUST continue processing as normal (as defined by OAuth 2.0 [RFC6749]). If the values are not equal, an error response indicating "invalid_grant" as described in Section 5.2 of [RFC6749] MUST be returned.

5. Compatibility

Server implementations of this specification MAY accept OAuth2.0 clients that do not implement this extension. If the "code_verifier" is not received from the client in the Authorization Request, servers supporting backwards compatibility revert to the OAuth 2.0 [RFC6749] protocol without this extension.

As the OAuth 2.0 [RFC6749] server responses are unchanged by this specification, client implementations of this specification do not need to know if the server has implemented this specification or not and SHOULD send the additional parameters as defined in Section 4 to all servers.

6. IANA Considerations

IANA has made the following registrations per this document.

6.1. OAuth Parameters Registry

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

- Parameter name: code_verifier
- Parameter usage location: token request
- Change controller: IESG
- Specification document(s): RFC 7636 (this document)

- Parameter name: code_challenge
- Parameter usage location: authorization request
- Change controller: IESG
- Specification document(s): RFC 7636 (this document)

- Parameter name: code_challenge_method
- Parameter usage location: authorization request
- Change controller: IESG
- Specification document(s): RFC 7636 (this document)

6.2. PKCE Code Challenge Method Registry

This specification establishes the "PKCE Code Challenge Methods" registry. The new registry should be a sub-registry of the "OAuth Parameters" registry.

Additional "code_challenge_method" types for use with the authorization endpoint are registered using the Specification Required policy [RFC5226], which includes review of the request by one or more Designated Experts (DEs). The DEs will ensure that there

is at least a two-week review of the request on the oauth-ext-review@ietf.org mailing list and that any discussion on that list converges before they respond to the request. To allow for the allocation of values prior to publication, the Designated Expert(s) may approve registration once they are satisfied that an acceptable specification will be published.

Registration requests and discussion on the oauth-ext-review@ietf.org mailing list should use an appropriate subject, such as "Request for PKCE code_challenge_method: example").

The Designated Expert(s) should consider the discussion on the mailing list, as well as the overall security properties of the challenge method when evaluating registration requests. New methods should not disclose the value of the code_verifier in the request to the Authorization endpoint. Denials should include an explanation and, if applicable, suggestions as to how to make the request successful.

6.2.1. Registration Template

Code Challenge Method Parameter Name:
: The name requested (e.g., "example"). Because a core goal of this specification is for the resulting representations to be compact, it is RECOMMENDED that the name be short -- not to exceed 8 characters without a compelling reason to do so. This name is case-sensitive. Names may not match other registered names in a case-insensitive manner unless the Designated Expert(s) states that there is a compelling reason to allow an exception in this particular case.

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

Specification Document(s):
: Reference to the document(s) that specifies the parameter, preferably including URI(s) that can be used to retrieve copies of the document(s). An indication of the relevant sections may also be included but is not required.

6.2.2. Initial Registry Contents

Per this document, IANA has registered the Code Challenge Method Parameter Names defined in Section 4.2 in this registry.

- o Code Challenge Method Parameter Name: plain
- o Change Controller: IESG
- o Specification Document(s): Section 4.2 of RFC 7636 (this document)

- o Code Challenge Method Parameter Name: S256
- o Change Controller: IESG
- o Specification Document(s): Section 4.2 of RFC 7636 (this document)

7. Security Considerations

7.1. Entropy of the code_verifier

The security model relies on the fact that the code verifier is not learned or guessed by the attacker. It is vitally important to adhere to this principle. As such, the code verifier has to be created in such a manner that it is cryptographically random and has high entropy that it is not practical for the attacker to guess.

The client SHOULD create a "code_verifier" with a minimum of 256 bits of entropy. This can be done by having a suitable random number generator create a 32-octet sequence. The octet sequence can then be base64url-encoded to produce a 43-octet URL safe string to use as a "code_challenge" that has the required entropy.

7.2. Protection against Eavesdroppers

Clients MUST NOT downgrade to "plain" after trying the "S256" method. Servers that support PKCE are required to support "S256", and servers that do not support PKCE will simply ignore the unknown "code_verifier". Because of this, an error when "S256" is presented can only mean that the server is faulty or that a MITM attacker is trying a downgrade attack.

The "S256" method protects against eavesdroppers observing or intercepting the "code_challenge", because the challenge cannot be used without the verifier. With the "plain" method, there is a chance that "code_challenge" will be observed by the attacker on the device or in the http request. Since the code challenge is the same as the code verifier in this case, the "plain" method does not protect against the eavesdropping of the initial request.

The use of "S256" protects against disclosure of the "code_verifier" value to an attacker.

Because of this, "plain" SHOULD NOT be used and exists only for compatibility with deployed implementations where the request path is already protected. The "plain" method SHOULD NOT be used in new implementations, unless they cannot support "S256" for some technical reason.

The "S256" code challenge method or other cryptographically secure code challenge method extension SHOULD be used. The "plain" code challenge method relies on the operating system and transport security not to disclose the request to an attacker.

If the code challenge method is "plain" and the code challenge is to be returned inside authorization "code" to achieve a stateless server, it MUST be encrypted in such a manner that only the server can decrypt and extract it.

7.3. Salting the code_challenge

To reduce implementation complexity, salting is not used in the production of the code challenge, as the code verifier contains sufficient entropy to prevent brute-force attacks. Concatenating a publicly known value to a code verifier (containing 256 bits of entropy) and then hashing it with SHA256 to produce a code challenge would not increase the number of attempts necessary to brute force a valid value for code verifier.

While the "S256" transformation is like hashing a password, there are important differences. Passwords tend to be relatively low-entropy words that can be hashed offline and the hash looked up in a dictionary. By concatenating a unique though public value to each password prior to hashing, the dictionary space that an attacker needs to search is greatly expanded.

Modern graphics processors now allow attackers to calculate hashes in real time faster than they could be looked up from a disk. This eliminates the value of the salt in increasing the complexity of a brute-force attack for even low-entropy passwords.

7.4. OAuth Security Considerations

All the OAuth security analysis presented in [RFC6819] applies, so readers SHOULD carefully follow it.

7.5. TLS Security Considerations

Current security considerations can be found in "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)" [BCP195]. This supersedes the TLS version recommendations in OAuth 2.0 [RFC6749].

8. References

8.1. Normative References

[BCP195] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", BCP 195, RFC 7525, May 2015, .

[RFC20] Cerf, V., "ASCII format for network interchange", STD 80, RFC 20, DOI 10.17487/RFC0020, October 1969, .

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, .

[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, .

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, .

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, DOI 10.17487/RFC5226, May 2008, .

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, .

[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, .

[RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, .

8.2. Informative References

[RFC6819] Lodderstedt, T., Ed., McGloin, M., and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations", RFC 6819, DOI 10.17487/RFC6819, January 2013, .

Appendix A. Notes on Implementing Base64url Encoding without Padding

This appendix describes how to implement a base64url-encoding function without padding, based upon the standard base64-encoding function that uses padding.

To be concrete, example C# code implementing these functions is shown below. Similar code could be used in other languages.

```
static string base64urlencode(byte [] arg)
{
  string s = Convert.ToBase64String(arg); // Regular base64 encoder
  s = s.Split('=')[0]; // Remove any trailing '='s
  s = s.Replace('+', '-'); // 62nd char of encoding
  s = s.Replace('/', '_'); // 63rd char of encoding
  return s;
}
```

An example correspondence between unencoded and encoded values follows. The octet sequence below encodes into the string below, which when decoded, reproduces the octet sequence.

3 236 255 224 193

A-z_4ME

Appendix B. Example for the S256 code_challenge_method

The client uses output of a suitable random number generator to create a 32-octet sequence. The octets representing the value in this example (using JSON array notation) are:

```
[116, 24, 223, 180, 151, 153, 224, 37, 79, 250, 96, 125, 216, 173,
187, 186, 22, 212, 37, 77, 105, 214, 191, 240, 91, 88, 5, 88, 83,
132, 141, 121]
```

Encoding this octet sequence as base64url provides the value of the code_verifier:

```
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```

The code_verifier is then hashed via the SHA256 hash function to produce:

```
[19, 211, 30, 150, 26, 26, 216, 236, 47, 22, 177, 12, 76, 152, 46,
 8, 118, 168, 120, 173, 109, 241, 68, 86, 110, 225, 137, 74, 203,
 112, 249, 195]
```

Encoding this octet sequence as base64url provides the value of the code_challenge:

```
E9Melhoa2OwvFrEMTJguCHaoeK1t8URWbuGJSstw-cM
```

The authorization request includes:

```
code_challenge=E9Melhoa2OwvFrEMTJguCHaoeK1t8URWbuGJSstw-cM
&code_challenge_method=S256
```

The authorization server then records the code_challenge and code_challenge_method along with the code that is granted to the client.

In the request to the token_endpoint, the client includes the code received in the authorization response as well as the additional parameter:

```
code_verifier=dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```

The authorization server retrieves the information for the code grant. Based on the recorded code_challenge_method being S256, it then hashes and base64url-encodes the value of code_verifier:

```
BASE64URL-ENCODE(SHA256(ASCII(code_verifier)))
```

The calculated value is then compared with the value of "code_challenge":

```
BASE64URL-ENCODE(SHA256(ASCII(code_verifier))) == code_challenge
```

If the two values are equal, then the authorization server can provide the tokens as long as there are no other errors in the request. If the values are not equal, then the request must be rejected, and an error returned.

Acknowledgements

The initial draft version of this specification was created by the OpenID AB/Connect Working Group of the OpenID Foundation.

This specification is the work of the OAuth Working Group, which includes dozens of active and dedicated participants. In particular, the following individuals contributed ideas, feedback, and wording that shaped and formed the final specification:

Anthony Nadalin, Microsoft
Axel Nenker, Deutsche Telekom
Breno de Medeiros, Google
Brian Campbell, Ping Identity
Chuck Mortimore, Salesforce
Dirk Balfanz, Google
Eduardo Gueiros, Jive Communications
Hannes Tschonfenig, ARM
James Manger, Telstra
Justin Richer, MIT Kerberos
Josh Mandel, Boston Children's Hospital
Lewis Adam, Motorola Solutions
Madjid Nakhjiri, Samsung
Michael B. Jones, Microsoft
Paul Madsen, Ping Identity
Phil Hunt, Oracle
Prateek Mishra, Oracle
Ryo Ito, mixi
Scott Tomilson, Ping Identity
Sergey Beryozkin
Takamichi Saito
Torsten Lodderstedt, Deutsche Telekom
William Denniss, Google

Authors' Addresses

Nat Sakimura (editor)
Nomura Research Institute
1-6-5 Marunouchi, Marunouchi Kitaguchi Bldg.
Chiyoda-ku, Tokyo 100-0005
Japan

Phone: +81-3-5533-2111
Email: n-sakimura@nri.co.jp
URI: http://nat.sakimura.org/

John Bradley
Ping Identity
Casilla 177, Sucursal Talagante
Talagante, RM
Chile

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

Naveen Agarwal
Google
1600 Amphitheatre Parkway
Mountain View, CA 94043
United States

Phone: +1 650-253-0000
Email: naa@google.com
URI: http://google.com/

Chapter 4

RFC 6819: OAuth 2.0 Threat Model and Security Considerations

The Threat Model and Security Considerations document was written to provide additional guidance beyond what is described in the core RFC. Much of this document was added after major providers had real implementation experience. The document describes many known attacks, either theoretical attacks or ones that have been demonstrated in the wild, and describes countermeasures for each.

If you are implementing an OAuth server from scratch, this is absolutely a must-read. Most of this advice is intended for implementers of authorization servers or resource servers, but some of it applies to client developers as well.

Internet Engineering Task Force (IETF)
Request for Comments: 6819
Category: Informational
ISSN: 2070-1721

T. Lodderstedt, Ed.
Deutsche Telekom AG
M. McGloin
IBM
P. Hunt
Oracle Corporation

January 2013

OAuth 2.0 Threat Model and Security Considerations

Abstract

This document gives additional security considerations for OAuth, beyond those in the OAuth 2.0 specification, based on a comprehensive threat model for the OAuth 2.0 protocol.

Status of This Memo

This document is not an Internet Standards Track specification; it is published for informational purposes.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Not all documents approved by the IESG are a candidate for any level of Internet Standard; see Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc6819.

Copyright Notice

Copyright (c) 2013 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 6
2. Overview 7
   2.1. Scope 7
   2.2. Attack Assumptions 7
   2.3. Architectural Assumptions 8
        2.3.1. Authorization Servers 8
        2.3.2. Resource Server 9
        2.3.3. Client 9
3. Security Features 9
   3.1. Tokens 10
        3.1.1. Scope 11
        3.1.2. Limited Access Token Lifetime 11
   3.2. Access Token 11
   3.3. Refresh Token 11
   3.4. Authorization "code" 12
   3.5. Redirect URI 13
   3.6. "state" Parameter 13
   3.7. Client Identifier 13
4. Threat Model 15
   4.1. Clients 16
        4.1.1. Threat: Obtaining Client Secrets 16
        4.1.2. Threat: Obtaining Refresh Tokens 17
        4.1.3. Threat: Obtaining Access Tokens 19
        4.1.4. Threat: End-User Credentials Phished Using
               Compromised or Embedded Browser 19
        4.1.5. Threat: Open Redirectors on Client 20
   4.2. Authorization Endpoint 21
        4.2.1. Threat: Password Phishing by Counterfeit
               Authorization Server 21
        4.2.2. Threat: User Unintentionally Grants Too
               Much Access Scope 21
        4.2.3. Threat: Malicious Client Obtains Existing
               Authorization by Fraud 22
        4.2.4. Threat: Open Redirector 22
   4.3. Token Endpoint 23
        4.3.1. Threat: Eavesdropping Access Tokens 23
        4.3.2. Threat: Obtaining Access Tokens from
               Authorization Server Database 23
        4.3.3. Threat: Disclosure of Client Credentials
               during Transmission 23
        4.3.4. Threat: Obtaining Client Secret from
               Authorization Server Database 24
        4.3.5. Threat: Obtaining Client Secret by Online Guessing .24

4.4. Obtaining Authorization 25
4.4.1. Authorization "code" 25
4.4.1.1. Threat: Eavesdropping or Leaking Authorization "codes" 25
4.4.1.2. Threat: Obtaining Authorization "codes" from Authorization Server Database 26
4.4.1.3. Threat: Online Guessing of Authorization "codes" 27
4.4.1.4. Threat: Malicious Client Obtains Authorization 27
4.4.1.5. Threat: Authorization "code" Phishing 29
4.4.1.6. Threat: User Session Impersonation 29
4.4.1.7. Threat: Authorization "code" Leakage through Counterfeit Client 30
4.4.1.8. Threat: CSRF Attack against redirect-uri ..32
4.4.1.9. Threat: Clickjacking Attack against Authorization 33
4.4.1.10. Threat: Resource Owner Impersonation 33
4.4.1.11. Threat: DoS Attacks That Exhaust Resources 34
4.4.1.12. Threat: DoS Using Manufactured Authorization "codes" 35
4.4.1.13. Threat: Code Substitution (OAuth Login) ..36
4.4.2. Implicit Grant 37
4.4.2.1. Threat: Access Token Leak in Transport/Endpoints 37
4.4.2.2. Threat: Access Token Leak in Browser History 38
4.4.2.3. Threat: Malicious Client Obtains Authorization 38
4.4.2.4. Threat: Manipulation of Scripts 38
4.4.2.5. Threat: CSRF Attack against redirect-uri ..39
4.4.2.6. Threat: Token Substitution (OAuth Login) ..39
4.4.3. Resource Owner Password Credentials 40
4.4.3.1. Threat: Accidental Exposure of Passwords at Client Site 41
4.4.3.2. Threat: Client Obtains Scopes without End-User Authorization 42
4.4.3.3. Threat: Client Obtains Refresh Token through Automatic Authorization 42
4.4.3.4. Threat: Obtaining User Passwords on Transport 43
4.4.3.5. Threat: Obtaining User Passwords from Authorization Server Database 43
4.4.3.6. Threat: Online Guessing 43
4.4.4. Client Credentials 44

```
      4.5. Refreshing an Access Token ................................44
           4.5.1. Threat: Eavesdropping Refresh Tokens from
                  Authorization Server ...............................44
           4.5.2. Threat: Obtaining Refresh Token from
                  Authorization Server Database ......................44
           4.5.3. Threat: Obtaining Refresh Token by Online
                  Guessing ...........................................45
           4.5.4. Threat: Refresh Token Phishing by
                  Counterfeit Authorization Server ...................45
      4.6. Accessing Protected Resources .............................46
           4.6.1. Threat: Eavesdropping Access Tokens on Transport ...46
           4.6.2. Threat: Replay of Authorized Resource
                  Server Requests ....................................46
           4.6.3. Threat: Guessing Access Tokens .....................46
           4.6.4. Threat: Access Token Phishing by
                  Counterfeit Resource Server ........................47
           4.6.5. Threat: Abuse of Token by Legitimate
                  Resource Server or Client ..........................48
           4.6.6. Threat: Leak of Confidential Data in HTTP Proxies ..48
           4.6.7. Threat: Token Leakage via Log Files and
                  HTTP Referrers .....................................48
   5. Security Considerations ........................................49
      5.1. General ...................................................49
           5.1.1. Ensure Confidentiality of Requests .................49
           5.1.2. Utilize Server Authentication ......................50
           5.1.3. Always Keep the Resource Owner Informed ............50
           5.1.4. Credentials ........................................51
                  5.1.4.1. Enforce Credential Storage
                           Protection Best Practices .................51
                  5.1.4.2. Online Attacks on Secrets .................52
           5.1.5. Tokens (Access, Refresh, Code) .....................53
                  5.1.5.1. Limit Token Scope .........................53
                  5.1.5.2. Determine Expiration Time .................54
                  5.1.5.3. Use Short Expiration Time .................54
                  5.1.5.4. Limit Number of Usages or One-Time Usage ..55
                  5.1.5.5. Bind Tokens to a Particular
                           Resource Server (Audience) ................55
                  5.1.5.6. Use Endpoint Address as Token Audience ....56
                  5.1.5.7. Use Explicitly Defined Scopes for
                           Audience and Tokens .......................56
                  5.1.5.8. Bind Token to Client id ...................56
                  5.1.5.9. Sign Self-Contained Tokens ................56
                  5.1.5.10. Encrypt Token Content ....................56
                  5.1.5.11. Adopt a Standard Assertion Format ........57
           5.1.6. Access Tokens ......................................57
```

5.2. Authorization Server 57
5.2.1. Authorization "codes" 57
5.2.1.1. Automatic Revocation of Derived Tokens If Abuse Is Detected 57
5.2.2. Refresh Tokens 57
5.2.2.1. Restricted Issuance of Refresh Tokens 57
5.2.2.2. Binding of Refresh Token to "client_id" ...58
5.2.2.3. Refresh Token Rotation 58
5.2.2.4. Revocation of Refresh Tokens 58
5.2.2.5. Device Identification 59
5.2.2.6. X-FRAME-OPTIONS Header 59
5.2.3. Client Authentication and Authorization 59
5.2.3.1. Don't Issue Secrets to Clients with Inappropriate Security Policy 60
5.2.3.2. Require User Consent for Public Clients without Secret 60
5.2.3.3. Issue a "client_id" Only in Combination with "redirect_uri" 61
5.2.3.4. Issue Installation-Specific Client Secrets 61
5.2.3.5. Validate Pre-Registered "redirect_uri" 62
5.2.3.6. Revoke Client Secrets 63
5.2.3.7. Use Strong Client Authentication (e.g., client_assertion/client_token) 63
5.2.4. End-User Authorization 63
5.2.4.1. Automatic Processing of Repeated Authorizations Requires Client Validation .63
5.2.4.2. Informed Decisions Based on Transparency ..63
5.2.4.3. Validation of Client Properties by End User 64
5.2.4.4. Binding of Authorization "code" to "client_id" 64
5.2.4.5. Binding of Authorization "code" to "redirect_uri" 64
5.3. Client App Security 65
5.3.1. Don't Store Credentials in Code or Resources Bundled with Software Packages 65
5.3.2. Use Standard Web Server Protection Measures (for Config Files and Databases) 65
5.3.3. Store Secrets in Secure Storage 65
5.3.4. Utilize Device Lock to Prevent Unauthorized Device Access 66
5.3.5. Link the "state" Parameter to User Agent Session ...66
5.4. Resource Servers 66
5.4.1. Authorization Headers 66
5.4.2. Authenticated Requests 67
5.4.3. Signed Requests 67
5.5. A Word on User Interaction and User-Installed Apps 68

6. Acknowledgements 69
7. References 69
7.1. Normative References 69
7.2. Informative References 69

1. Introduction

This document gives additional security considerations for OAuth, beyond those in the OAuth specification, based on a comprehensive threat model for the OAuth 2.0 protocol [RFC6749]. It contains the following content:

- Documents any assumptions and scope considered when creating the threat model.
- Describes the security features built into the OAuth protocol and how they are intended to thwart attacks.
- Gives a comprehensive threat model for OAuth and describes the respective countermeasures to thwart those threats.

Threats include any intentional attacks on OAuth tokens and resources protected by OAuth tokens, as well as security risks introduced if the proper security measures are not put in place. Threats are structured along the lines of the protocol structure to help development teams implement each part of the protocol securely, for example, all threats for granting access, or all threats for a particular grant type, or all threats for protecting the resource server.

Note: This document cannot assess the probability or the risk associated with a particular threat because those aspects strongly depend on the particular application and deployment OAuth is used to protect. Similarly, impacts are given on a rather abstract level. But the information given here may serve as a foundation for deployment-specific threat models. Implementors may refine and detail the abstract threat model in order to account for the specific properties of their deployment and to come up with a risk analysis. As this document is based on the base OAuth 2.0 specification, it does not consider proposed extensions such as client registration or discovery, many of which are still under discussion.

2. Overview

2.1. Scope

This security considerations document only considers clients bound to a particular deployment as supported by [RFC6749]. Such deployments have the following characteristics:

- Resource server URLs are static and well-known at development time; authorization server URLs can be static or discovered.
- Token scope values (e.g., applicable URLs and methods) are well-known at development time.
- Client registration is out of scope of the current core specification. Therefore, this document assumes a broad variety of options, from static registration during development time to dynamic registration at runtime.

The following are considered out of scope:

- Communication between the authorization server and resource server.
- Token formats.
- Except for the resource owner password credentials grant type (see [RFC6749], Section 4.3), the mechanism used by authorization servers to authenticate the user.
- Mechanism by which a user obtained an assertion and any resulting attacks mounted as a result of the assertion being false.
- Clients not bound to a specific deployment: An example could be a mail client with support for contact list access via the portable contacts API (see [Portable-Contacts]). Such clients cannot be registered upfront with a particular deployment and should dynamically discover the URLs relevant for the OAuth protocol.

2.2. Attack Assumptions

The following assumptions relate to an attacker and resources available to an attacker. It is assumed that:

- the attacker has full access to the network between the client and authorization servers and the client and the resource server, respectively. The attacker may eavesdrop on any communications

between those parties. He is not assumed to have access to communication between the authorization server and resource server.

- an attacker has unlimited resources to mount an attack.

- two of the three parties involved in the OAuth protocol may collude to mount an attack against the 3rd party. For example, the client and authorization server may be under control of an attacker and collude to trick a user to gain access to resources.

2.3. Architectural Assumptions

This section documents assumptions about the features, limitations, and design options of the different entities of an OAuth deployment along with the security-sensitive data elements managed by those entities. These assumptions are the foundation of the threat analysis.

The OAuth protocol leaves deployments with a certain degree of freedom regarding how to implement and apply the standard. The core specification defines the core concepts of an authorization server and a resource server. Both servers can be implemented in the same server entity, or they may also be different entities. The latter is typically the case for multi-service providers with a single authentication and authorization system and is more typical in middleware architectures.

2.3.1. Authorization Servers

The following data elements are stored or accessible on the authorization server:

- usernames and passwords

- client ids and secrets

- client-specific refresh tokens

- client-specific access tokens (in the case of handle-based design; see Section 3.1)

- HTTPS certificate/key

- per-authorization process (in the case of handle-based design; Section 3.1): "redirect_uri", "client_id", authorization "code"

2.3.2. Resource Server

The following data elements are stored or accessible on the resource server:

- o user data (out of scope)
- o HTTPS certificate/key
- o either authorization server credentials (handle-based design; see Section 3.1) or authorization server shared secret/public key (assertion-based design; see Section 3.1)
- o access tokens (per request)

It is assumed that a resource server has no knowledge of refresh tokens, user passwords, or client secrets.

2.3.3. Client

In OAuth, a client is an application making protected resource requests on behalf of the resource owner and with its authorization. There are different types of clients with different implementation and security characteristics, such as web, user-agent-based, and native applications. A full definition of the different client types and profiles is given in [RFC6749], Section 2.1.

The following data elements are stored or accessible on the client:

- o client id (and client secret or corresponding client credential)
- o one or more refresh tokens (persistent) and access tokens (transient) per end user or other security-context or delegation context
- o trusted certification authority (CA) certificates (HTTPS)
- o per-authorization process: "redirect_uri", authorization "code"

3. Security Features

These are some of the security features that have been built into the OAuth 2.0 protocol to mitigate attacks and security issues.

3.1. Tokens

OAuth makes extensive use of many kinds of tokens (access tokens, refresh tokens, authorization "codes"). The information content of a token can be represented in two ways, as follows:

Handle (or artifact) A 'handle' is a reference to some internal data structure within the authorization server; the internal data structure contains the attributes of the token, such as user id (UID), scope, etc. Handles enable simple revocation and do not require cryptographic mechanisms to protect token content from being modified. On the other hand, handles require communication between the issuing and consuming entity (e.g., the authorization server and resource server) in order to validate the token and obtain token-bound data. This communication might have a negative impact on performance and scalability if both entities reside on different systems. Handles are therefore typically used if the issuing and consuming entity are the same. A 'handle' token is often referred to as an 'opaque' token because the resource server does not need to be able to interpret the token directly; it simply uses the token.

Assertion (aka self-contained token) An assertion is a parseable token. An assertion typically has a duration, has an audience, and is digitally signed in order to ensure data integrity and origin authentication. It contains information about the user and the client. Examples of assertion formats are Security Assertion Markup Language (SAML) assertions [OASIS.saml-core-2.0-os] and Kerberos tickets [RFC4120]. Assertions can typically be directly validated and used by a resource server without interactions with the authorization server. This results in better performance and scalability in deployments where the issuing and consuming entities reside on different systems. Implementing token revocation is more difficult with assertions than with handles.

Tokens can be used in two ways to invoke requests on resource servers, as follows:

bearer token A 'bearer token' is a token that can be used by any client who has received the token (e.g., [RFC6750]). Because mere possession is enough to use the token, it is important that communication between endpoints be secured to ensure that only authorized endpoints may capture the token. The bearer token is convenient for client applications, as it does not require them to do anything to use them (such as a proof of identity). Bearer tokens have similar characteristics to web single-sign-on (SSO) cookies used in browsers.

proof token A 'proof token' is a token that can only be used by a specific client. Each use of the token requires the client to perform some action that proves that it is the authorized user of the token. Examples of this are MAC-type access tokens, which require the client to digitally sign the resource request with a secret corresponding to the particular token sent with the request (e.g., [OAuth-HTTP-MAC]).

3.1.1. Scope

A scope represents the access authorization associated with a particular token with respect to resource servers, resources, and methods on those resources. Scopes are the OAuth way to explicitly manage the power associated with an access token. A scope can be controlled by the authorization server and/or the end user in order to limit access to resources for OAuth clients that these parties deem less secure or trustworthy. Optionally, the client can request the scope to apply to the token but only for a lesser scope than would otherwise be granted, e.g., to reduce the potential impact if this token is sent over non-secure channels. A scope is typically complemented by a restriction on a token's lifetime.

3.1.2. Limited Access Token Lifetime

The protocol parameter "expires_in" allows an authorization server (based on its policies or on behalf of the end user) to limit the lifetime of an access token and to pass this information to the client. This mechanism can be used to issue short-lived tokens to OAuth clients that the authorization server deems less secure, or where sending tokens over non-secure channels.

3.2. Access Token

An access token is used by a client to access a resource. Access tokens typically have short life spans (minutes or hours) that cover typical session lifetimes. An access token may be refreshed through the use of a refresh token. The short lifespan of an access token, in combination with the usage of refresh tokens, enables the possibility of passive revocation of access authorization on the expiry of the current access token.

3.3. Refresh Token

A refresh token represents a long-lasting authorization of a certain client to access resources on behalf of a resource owner. Such tokens are exchanged between the client and authorization server only. Clients use this kind of token to obtain ("refresh") new access tokens used for resource server invocations.

A refresh token, coupled with a short access token lifetime, can be used to grant longer access to resources without involving end-user authorization. This offers an advantage where resource servers and authorization servers are not the same entity, e.g., in a distributed environment, as the refresh token is always exchanged at the authorization server. The authorization server can revoke the refresh token at any time, causing the granted access to be revoked once the current access token expires. Because of this, a short access token lifetime is important if timely revocation is a high priority.

The refresh token is also a secret bound to the client identifier and client instance that originally requested the authorization; the refresh token also represents the original resource owner grant. This is ensured by the authorization process as follows:

1. The resource owner and user agent safely deliver the authorization "code" to the client instance in the first place.

2. The client uses it immediately in secure transport-level communications to the authorization server and then securely stores the long-lived refresh token.

3. The client always uses the refresh token in secure transport-level communications to the authorization server to get an access token (and optionally roll over the refresh token).

So, as long as the confidentiality of the particular token can be ensured by the client, a refresh token can also be used as an alternative means to authenticate the client instance itself.

3.4. Authorization "code"

An authorization "code" represents the intermediate result of a successful end-user authorization process and is used by the client to obtain access and refresh tokens. Authorization "codes" are sent to the client's redirect URI instead of tokens for two purposes:

1. Browser-based flows expose protocol parameters to potential attackers via URI query parameters (HTTP referrer), the browser cache, or log file entries, and could be replayed. In order to reduce this threat, short-lived authorization "codes" are passed instead of tokens and exchanged for tokens over a more secure direct connection between the client and the authorization server.

2. It is much simpler to authenticate clients during the direct request between the client and the authorization server than in the context of the indirect authorization request. The latter would require digital signatures.

3.5. Redirect URI

A redirect URI helps to detect malicious clients and prevents phishing attacks from clients attempting to trick the user into believing the phisher is the client. The value of the actual redirect URI used in the authorization request has to be presented and is verified when an authorization "code" is exchanged for tokens. This helps to prevent attacks where the authorization "code" is revealed through redirectors and counterfeit web application clients. The authorization server should require public clients and confidential clients using the implicit grant type to pre-register their redirect URIs and validate against the registered redirect URI in the authorization request.

3.6. "state" Parameter

The "state" parameter is used to link requests and callbacks to prevent cross-site request forgery attacks (see Section 4.4.1.8) where an attacker authorizes access to his own resources and then tricks a user into following a redirect with the attacker's token. This parameter should bind to the authenticated state in a user agent and, as per the core OAuth spec, the user agent must be capable of keeping it in a location accessible only by the client and user agent, i.e., protected by same-origin policy.

3.7. Client Identifier

Authentication protocols have typically not taken into account the identity of the software component acting on behalf of the end user. OAuth does this in order to increase the security level in delegated authorization scenarios and because the client will be able to act without the user being present.

OAuth uses the client identifier to collate associated requests to the same originator, such as

- a particular end-user authorization process and the corresponding request on the token's endpoint to exchange the authorization "code" for tokens, or

o the initial authorization and issuance of a token by an end user to a particular client, and subsequent requests by this client to obtain tokens without user consent (automatic processing of repeated authorizations)

This identifier may also be used by the authorization server to display relevant registration information to a user when requesting consent for a scope requested by a particular client. The client identifier may be used to limit the number of requests for a particular client or to charge the client per request. It may furthermore be useful to differentiate access by different clients, e.g., in server log files.

OAuth defines two client types, confidential and public, based on their ability to authenticate with the authorization server (i.e., ability to maintain the confidentiality of their client credentials). Confidential clients are capable of maintaining the confidentiality of client credentials (i.e., a client secret associated with the client identifier) or capable of secure client authentication using other means, such as a client assertion (e.g., SAML) or key cryptography. The latter is considered more secure.

The authorization server should determine whether the client is capable of keeping its secret confidential or using secure authentication. Alternatively, the end user can verify the identity of the client, e.g., by only installing trusted applications. The redirect URI can be used to prevent the delivery of credentials to a counterfeit client after obtaining end-user authorization in some cases but can't be used to verify the client identifier.

Clients can be categorized as follows based on the client type, profile (e.g., native vs. web application; see [RFC6749], Section 9), and deployment model:

Deployment-independent "client_id" with pre-registered "redirect_uri" and without "client_secret" Such an identifier is used by multiple installations of the same software package. The identifier of such a client can only be validated with the help of the end-user. This is a viable option for native applications in order to identify the client for the purpose of displaying meta information about the client to the user and to differentiate clients in log files. Revocation of the rights associated with such a client identifier will affect ALL deployments of the respective software.

Deployment-independent "client_id" with pre-registered "redirect_uri" and with "client_secret" This is an option for native applications only, since web applications would require different redirect URIs. This category is not advisable because the client secret cannot be protected appropriately (see Section 4.1.1). Due to its security weaknesses, such client identities have the same trust level as deployment-independent clients without secrets. Revocation will affect ALL deployments.

Deployment-specific "client_id" with pre-registered "redirect_uri" and with "client_secret" The client registration process ensures the validation of the client's properties, such as redirect URI, web site URL, web site name, and contacts. Such a client identifier can be utilized for all relevant use cases cited above. This level can be achieved for web applications in combination with a manual or user-bound registration process. Achieving this level for native applications is much more difficult. Either the installation of the application is conducted by an administrator, who validates the client's authenticity, or the process from validating the application to the installation of the application on the device and the creation of the client credentials is controlled end-to-end by a single entity (e.g., application market provider). Revocation will affect a single deployment only.

Deployment-specific "client_id" with "client_secret" without validated properties Such a client can be recognized by the authorization server in transactions with subsequent requests (e.g., authorization and token issuance, refresh token issuance, and access token refreshment). The authorization server cannot assure any property of the client to end users. Automatic processing of re-authorizations could be allowed as well. Such client credentials can be generated automatically without any validation of client properties, which makes it another option, especially for native applications. Revocation will affect a single deployment only.

4. Threat Model

This section gives a comprehensive threat model of OAuth 2.0. Threats are grouped first by attacks directed against an OAuth component, which are the client, authorization server, and resource server. Subsequently, they are grouped by flow, e.g., obtain token or access protected resources. Every countermeasure description refers to a detailed description in Section 5.

4.1. Clients

This section describes possible threats directed to OAuth clients.

4.1.1. Threat: Obtaining Client Secrets

The attacker could try to get access to the secret of a particular client in order to:

- o replay its refresh tokens and authorization "codes", or

- o obtain tokens on behalf of the attacked client with the privileges of that "client_id" acting as an instance of the client.

The resulting impact would be the following:

- o Client authentication of access to the authorization server can be bypassed.

- o Stolen refresh tokens or authorization "codes" can be replayed.

Depending on the client category, the following attacks could be utilized to obtain the client secret.

Attack: Obtain Secret From Source Code or Binary:

This applies for all client types. For open source projects, secrets can be extracted directly from source code in their public repositories. Secrets can be extracted from application binaries just as easily when the published source is not available to the attacker. Even if an application takes significant measures to obfuscate secrets in their application distribution, one should consider that the secret can still be reverse-engineered by anyone with access to a complete functioning application bundle or binary.

Countermeasures:

- o Don't issue secrets to public clients or clients with inappropriate security policy (Section 5.2.3.1).

- o Require user consent for public clients (Section 5.2.3.2).

- o Use deployment-specific client secrets (Section 5.2.3.4).

- o Revoke client secrets (Section 5.2.3.6).

Attack: Obtain a Deployment-Specific Secret:

An attacker may try to obtain the secret from a client installation, either from a web site (web server) or a particular device (native application).

Countermeasures:

- Web server: Apply standard web server protection measures (for config files and databases) (see Section 5.3.2).
- Native applications: Store secrets in secure local storage (Section 5.3.3).
- Revoke client secrets (Section 5.2.3.6).

4.1.2. Threat: Obtaining Refresh Tokens

Depending on the client type, there are different ways that refresh tokens may be revealed to an attacker. The following sub-sections give a more detailed description of the different attacks with respect to different client types and further specialized countermeasures. Before detailing those threats, here are some generally applicable countermeasures:

- The authorization server should validate the client id associated with the particular refresh token with every refresh request (Section 5.2.2.2).
- Limit token scope (Section 5.1.5.1).
- Revoke refresh tokens (Section 5.2.2.4).
- Revoke client secrets (Section 5.2.3.6).
- Refresh tokens can automatically be replaced in order to detect unauthorized token usage by another party (see "Refresh Token Rotation", Section 5.2.2.3).

Attack: Obtain Refresh Token from Web Application:

An attacker may obtain the refresh tokens issued to a web application by way of overcoming the web server's security controls.

Impact: Since a web application manages the user accounts of a certain site, such an attack would result in an exposure of all refresh tokens on that site to the attacker.

Countermeasures:

- o Standard web server protection measures (Section 5.3.2).

- o Use strong client authentication (e.g., client_assertion/ client_token) so the attacker cannot obtain the client secret required to exchange the tokens (Section 5.2.3.7).

Attack: Obtain Refresh Token from Native Clients:

On native clients, leakage of a refresh token typically affects a single user only.

Read from local file system: The attacker could try to get file system access on the device and read the refresh tokens. The attacker could utilize a malicious application for that purpose.

Countermeasures:

- o Store secrets in secure storage (Section 5.3.3).

- o Utilize device lock to prevent unauthorized device access (Section 5.3.4).

Attack: Steal Device:

The host device (e.g., mobile phone) may be stolen. In that case, the attacker gets access to all applications under the identity of the legitimate user.

Countermeasures:

- o Utilize device lock to prevent unauthorized device access (Section 5.3.4).

- o Where a user knows the device has been stolen, they can revoke the affected tokens (Section 5.2.2.4).

Attack: Clone Device:

All device data and applications are copied to another device. Applications are used as-is on the target device.

Countermeasures:

- o Utilize device lock to prevent unauthorized device access (Section 5.3.4).
- o Combine refresh token request with device identification (Section 5.2.2.5).
- o Refresh token rotation (Section 5.2.2.3).
- o Where a user knows the device has been cloned, they can use refresh token revocation (Section 5.2.2.4).

4.1.3. Threat: Obtaining Access Tokens

Depending on the client type, there are different ways that access tokens may be revealed to an attacker. Access tokens could be stolen from the device if the application stores them in a storage device that is accessible to other applications.

Impact: Where the token is a bearer token and no additional mechanism is used to identify the client, the attacker can access all resources associated with the token and its scope.

Countermeasures:

- o Keep access tokens in transient memory and limit grants (Section 5.1.6).
- o Limit token scope (Section 5.1.5.1).
- o Keep access tokens in private memory or apply same protection means as for refresh tokens (Section 5.2.2).
- o Keep access token lifetime short (Section 5.1.5.3).

4.1.4. Threat: End-User Credentials Phished Using Compromised or Embedded Browser

A malicious application could attempt to phish end-user passwords by misusing an embedded browser in the end-user authorization process, or by presenting its own user interface instead of allowing a trusted system browser to render the authorization user interface. By doing so, the usual visual trust mechanisms may be bypassed (e.g., Transport Layer Security (TLS) confirmation, web site mechanisms). By using an embedded or internal client application user interface, the client application has access to additional information to which it should not have access (e.g., UID/password).

Impact: If the client application or the communication is compromised, the user would not be aware of this, and all information in the authorization exchange, such as username and password, could be captured.

Countermeasures:

- o The OAuth flow is designed so that client applications never need to know user passwords. Client applications should avoid directly asking users for their credentials. In addition, end users could be educated about phishing attacks and best practices, such as only accessing trusted clients, as OAuth does not provide any protection against malicious applications and the end user is solely responsible for the trustworthiness of any native application installed.

- o Client applications could be validated prior to publication in an application market for users to access. That validation is out of scope for OAuth but could include validating that the client application handles user authentication in an appropriate way.

- o Client developers should not write client applications that collect authentication information directly from users and should instead delegate this task to a trusted system component, e.g., the system browser.

4.1.5. Threat: Open Redirectors on Client

An open redirector is an endpoint using a parameter to automatically redirect a user agent to the location specified by the parameter value without any validation. If the authorization server allows the client to register only part of the redirect URI, an attacker can use an open redirector operated by the client to construct a redirect URI that will pass the authorization server validation but will send the authorization "code" or access token to an endpoint under the control of the attacker.

Impact: An attacker could gain access to authorization "codes" or access tokens.

Countermeasures:

- o Require clients to register full redirect URI (Section 5.2.3.5).

4.2. Authorization Endpoint

4.2.1. Threat: Password Phishing by Counterfeit Authorization Server

OAuth makes no attempt to verify the authenticity of the authorization server. A hostile party could take advantage of this by intercepting the client's requests and returning misleading or otherwise incorrect responses. This could be achieved using DNS or Address Resolution Protocol (ARP) spoofing. Wide deployment of OAuth and similar protocols may cause users to become inured to the practice of being redirected to web sites where they are asked to enter their passwords. If users are not careful to verify the authenticity of these web sites before entering their credentials, it will be possible for attackers to exploit this practice to steal users' passwords.

Countermeasures:

- Authorization servers should consider such attacks when developing services based on OAuth and should require the use of transport-layer security for any requests where the authenticity of the authorization server or of request responses is an issue (see Section 5.1.2).

- Authorization servers should attempt to educate users about the risks posed by phishing attacks and should provide mechanisms that make it easy for users to confirm the authenticity of their sites.

4.2.2. Threat: User Unintentionally Grants Too Much Access Scope

When obtaining end-user authorization, the end user may not understand the scope of the access being granted and to whom, or they may end up providing a client with access to resources that should not be permitted.

Countermeasures:

- Explain the scope (resources and the permissions) the user is about to grant in an understandable way (Section 5.2.4.2).

- Narrow the scope, based on the client. When obtaining end-user authorization and where the client requests scope, the authorization server may want to consider whether to honor that scope based on the client identifier. That decision is between the client and authorization server and is outside the scope of this spec. The authorization server may also want to consider what scope to grant based on the client type, e.g., providing lower scope to public clients (Section 5.1.5.1).

4.2.3. Threat: Malicious Client Obtains Existing Authorization by Fraud

Authorization servers may wish to automatically process authorization requests from clients that have been previously authorized by the user. When the user is redirected to the authorization server's end-user authorization endpoint to grant access, the authorization server detects that the user has already granted access to that particular client. Instead of prompting the user for approval, the authorization server automatically redirects the user back to the client.

A malicious client may exploit that feature and try to obtain such an authorization "code" instead of the legitimate client.

Countermeasures:

- o Authorization servers should not automatically process repeat authorizations to public clients unless the client is validated using a pre-registered redirect URI (Section 5.2.3.5).

- o Authorization servers can mitigate the risks associated with automatic processing by limiting the scope of access tokens obtained through automated approvals (Section 5.1.5.1).

4.2.4. Threat: Open Redirector

An attacker could use the end-user authorization endpoint and the redirect URI parameter to abuse the authorization server as an open redirector. An open redirector is an endpoint using a parameter to automatically redirect a user agent to the location specified by the parameter value without any validation.

Impact: An attacker could utilize a user's trust in an authorization server to launch a phishing attack.

Countermeasures:

- o Require clients to register any full redirect URIs (Section 5.2.3.5).

- o Don't redirect to a redirect URI if the client identifier or redirect URI can't be verified (Section 5.2.3.5).

4.3. Token Endpoint

4.3.1. Threat: Eavesdropping Access Tokens

Attackers may attempt to eavesdrop access tokens in transit from the authorization server to the client.

Impact: The attacker is able to access all resources with the permissions covered by the scope of the particular access token.

Countermeasures:

- As per the core OAuth spec, the authorization servers must ensure that these transmissions are protected using transport-layer mechanisms such as TLS (see Section 5.1.1).
- If end-to-end confidentiality cannot be guaranteed, reducing scope (see Section 5.1.5.1) and expiry time (Section 5.1.5.3) for access tokens can be used to reduce the damage in case of leaks.

4.3.2. Threat: Obtaining Access Tokens from Authorization Server Database

This threat is applicable if the authorization server stores access tokens as handles in a database. An attacker may obtain access tokens from the authorization server's database by gaining access to the database or launching a SQL injection attack.

Impact: Disclosure of all access tokens.

Countermeasures:

- Enforce system security measures (Section 5.1.4.1.1).
- Store access token hashes only (Section 5.1.4.1.3).
- Enforce standard SQL injection countermeasures (Section 5.1.4.1.2).

4.3.3. Threat: Disclosure of Client Credentials during Transmission

An attacker could attempt to eavesdrop the transmission of client credentials between the client and server during the client authentication process or during OAuth token requests.

Impact: Revelation of a client credential enabling phishing or impersonation of a client service.

Countermeasures:

- The transmission of client credentials must be protected using transport-layer mechanisms such as TLS (see Section 5.1.1).

- Use alternative authentication means that do not require the sending of plaintext credentials over the wire (e.g., Hash-based Message Authentication Code).

4.3.4. Threat: Obtaining Client Secret from Authorization Server Database

An attacker may obtain valid "client_id"/secret combinations from the authorization server's database by gaining access to the database or launching a SQL injection attack.

Impact: Disclosure of all "client_id"/secret combinations. This allows the attacker to act on behalf of legitimate clients.

Countermeasures:

- Enforce system security measures (Section 5.1.4.1.1).

- Enforce standard SQL injection countermeasures (Section 5.1.4.1.2).

- Ensure proper handling of credentials as per "Enforce Credential Storage Protection Best Practices" (Section 5.1.4.1).

4.3.5. Threat: Obtaining Client Secret by Online Guessing

An attacker may try to guess valid "client_id"/secret pairs.

Impact: Disclosure of a single "client_id"/secret pair.

Countermeasures:

- Use high entropy for secrets (Section 5.1.4.2.2).

- Lock accounts (Section 5.1.4.2.3).

- Use strong client authentication (Section 5.2.3.7).

4.4. Obtaining Authorization

This section covers threats that are specific to certain flows utilized to obtain access tokens. Each flow is characterized by response types and/or grant types on the end-user authorization and token endpoint, respectively.

4.4.1. Authorization "code"

4.4.1.1. Threat: Eavesdropping or Leaking Authorization "codes"

An attacker could try to eavesdrop transmission of the authorization "code" between the authorization server and client. Furthermore, authorization "codes" are passed via the browser, which may unintentionally leak those codes to untrusted web sites and attackers in different ways:

- o Referrer headers: Browsers frequently pass a "referer" header when a web page embeds content, or when a user travels from one web page to another web page. These referrer headers may be sent even when the origin site does not trust the destination site. The referrer header is commonly logged for traffic analysis purposes.
- o Request logs: Web server request logs commonly include query parameters on requests.
- o Open redirectors: Web sites sometimes need to send users to another destination via a redirector. Open redirectors pose a particular risk to web-based delegation protocols because the redirector can leak verification codes to untrusted destination sites.
- o Browser history: Web browsers commonly record visited URLs in the browser history. Another user of the same web browser may be able to view URLs that were visited by previous users.

Note: A description of similar attacks on the SAML protocol can be found at [OASIS.sstc-saml-bindings-1.1], Section 4.1.1.9.1; [Sec-Analysis]; and [OASIS.sstc-sec-analysis-response-01].

Countermeasures:

- As per the core OAuth spec, the authorization server as well as the client must ensure that these transmissions are protected using transport-layer mechanisms such as TLS (see Section 5.1.1).

- The authorization server will require the client to authenticate wherever possible, so the binding of the authorization "code" to a certain client can be validated in a reliable way (see Section 5.2.4.4).

- Use short expiry time for authorization "codes" (Section 5.1.5.3).

- The authorization server should enforce a one-time usage restriction (see Section 5.1.5.4).

- If an authorization server observes multiple attempts to redeem an authorization "code", the authorization server may want to revoke all tokens granted based on the authorization "code" (see Section 5.2.1.1).

- In the absence of these countermeasures, reducing scope (Section 5.1.5.1) and expiry time (Section 5.1.5.3) for access tokens can be used to reduce the damage in case of leaks.

- The client server may reload the target page of the redirect URI in order to automatically clean up the browser cache.

4.4.1.2. Threat: Obtaining Authorization "codes" from Authorization Server Database

This threat is applicable if the authorization server stores authorization "codes" as handles in a database. An attacker may obtain authorization "codes" from the authorization server's database by gaining access to the database or launching a SQL injection attack.

Impact: Disclosure of all authorization "codes", most likely along with the respective "redirect_uri" and "client_id" values.

Countermeasures:

- Best practices for credential storage protection should be employed (Section 5.1.4.1).

- Enforce system security measures (Section 5.1.4.1.1).

- o Store access token hashes only (Section 5.1.4.1.3).
- o Enforce standard SQL injection countermeasures (Section 5.1.4.1.2).

4.4.1.3. Threat: Online Guessing of Authorization "codes"

An attacker may try to guess valid authorization "code" values and send the guessed code value using the grant type "code" in order to obtain a valid access token.

Impact: Disclosure of a single access token and probably also an associated refresh token.

Countermeasures:

- o Handle-based tokens must use high entropy (Section 5.1.4.2.2).
- o Assertion-based tokens should be signed (Section 5.1.5.9).
- o Authenticate the client; this adds another value that the attacker has to guess (Section 5.2.3.4).
- o Bind the authorization "code" to the redirect URI; this adds another value that the attacker has to guess (Section 5.2.4.5).
- o Use short expiry time for tokens (Section 5.1.5.3).

4.4.1.4. Threat: Malicious Client Obtains Authorization

A malicious client could pretend to be a valid client and obtain an access authorization in this way. The malicious client could even utilize screen-scraping techniques in order to simulate a user's consent in the authorization flow.

Assumption: It is not the task of the authorization server to protect the end-user's device from malicious software. This is the responsibility of the platform running on the particular device, probably in cooperation with other components of the respective ecosystem (e.g., an application management infrastructure). The sole responsibility of the authorization server is to control access to the end-user's resources maintained in resource servers and to prevent unauthorized access to them via the OAuth protocol. Based on this assumption, the following countermeasures are available to cope with the threat.

Countermeasures:

- The authorization server should authenticate the client, if possible (see Section 5.2.3.4). Note: The authentication takes place after the end user has authorized the access.

- The authorization server should validate the client's redirect URI against the pre-registered redirect URI, if one exists (see Section 5.2.3.5). Note: An invalid redirect URI indicates an invalid client, whereas a valid redirect URI does not necessarily indicate a valid client. The level of confidence depends on the client type. For web applications, the level of confidence is high, since the redirect URI refers to the globally unique network endpoint of this application, whose fully qualified domain name (FQDN) is also validated using HTTPS server authentication by the user agent. In contrast, for native clients, the redirect URI typically refers to device local resources, e.g., a custom scheme. So, a malicious client on a particular device can use the valid redirect URI the legitimate client uses on all other devices.

- After authenticating the end user, the authorization server should ask him/her for consent. In this context, the authorization server should explain to the end user the purpose, scope, and duration of the authorization the client asked for. Moreover, the authorization server should show the user any identity information it has for that client. It is up to the user to validate the binding of this data to the particular application (e.g., Name) and to approve the authorization request (see Section 5.2.4.3).

- The authorization server should not perform automatic re-authorizations for clients it is unable to reliably authenticate or validate (see Section 5.2.4.1).

- If the authorization server automatically authenticates the end user, it may nevertheless require some user input in order to prevent screen scraping. Examples are CAPTCHAs (Completely Automated Public Turing tests to tell Computers and Humans Apart) or other multi-factor authentication techniques such as random questions, token code generators, etc.

- The authorization server may also limit the scope of tokens it issues to clients it cannot reliably authenticate (see Section 5.1.5.1).

4.4.1.5. Threat: Authorization "code" Phishing

A hostile party could impersonate the client site and get access to the authorization "code". This could be achieved using DNS or ARP spoofing. This applies to clients, which are web applications; thus, the redirect URI is not local to the host where the user's browser is running.

Impact: This affects web applications and may lead to a disclosure of authorization "codes" and, potentially, the corresponding access and refresh tokens.

Countermeasures:

It is strongly recommended that one of the following countermeasures be utilized in order to prevent this attack:

- The redirect URI of the client should point to an HTTPS-protected endpoint, and the browser should be utilized to authenticate this redirect URI using server authentication (see Section 5.1.2).

- The authorization server should require that the client be authenticated, i.e., confidential client, so the binding of the authorization "code" to a certain client can be validated in a reliable way (see Section 5.2.4.4).

4.4.1.6. Threat: User Session Impersonation

A hostile party could impersonate the client site and impersonate the user's session on this client. This could be achieved using DNS or ARP spoofing. This applies to clients, which are web applications; thus, the redirect URI is not local to the host where the user's browser is running.

Impact: An attacker who intercepts the authorization "code" as it is sent by the browser to the callback endpoint can gain access to protected resources by submitting the authorization "code" to the client. The client will exchange the authorization "code" for an access token and use the access token to access protected resources for the benefit of the attacker, delivering protected resources to the attacker, or modifying protected resources as directed by the attacker. If OAuth is used by the client to delegate authentication to a social site (e.g., as in the implementation of a "Login" button on a third-party social network site), the attacker can use the intercepted authorization "code" to log into the client as the user.

Note: Authenticating the client during authorization "code" exchange will not help to detect such an attack, as it is the legitimate client that obtains the tokens.

Countermeasures:

- In order to prevent an attacker from impersonating the end-user's session, the redirect URI of the client should point to an HTTPS protected endpoint, and the browser should be utilized to authenticate this redirect URI using server authentication (see Section 5.1.2).

4.4.1.7. Threat: Authorization "code" Leakage through Counterfeit Client

The attacker leverages the authorization "code" grant type in an attempt to get another user (victim) to log in, authorize access to his/her resources, and subsequently obtain the authorization "code" and inject it into a client application using the attacker's account. The goal is to associate an access authorization for resources of the victim with the user account of the attacker on a client site.

The attacker abuses an existing client application and combines it with his own counterfeit client web site. The attacker depends on the victim expecting the client application to request access to a certain resource server. The victim, seeing only a normal request from an expected application, approves the request. The attacker then uses the victim's authorization to gain access to the information unknowingly authorized by the victim.

The attacker conducts the following flow:

1. The attacker accesses the client web site (or application) and initiates data access to a particular resource server. The client web site in turn initiates an authorization request to the resource server's authorization server. Instead of proceeding with the authorization process, the attacker modifies the authorization server end-user authorization URL as constructed by the client to include a redirect URI parameter referring to a web site under his control (attacker's web site).

2. The attacker tricks another user (the victim) into opening that modified end-user authorization URI and authorizing access (e.g., via an email link or blog link). The way the attacker achieves this goal is out of scope.

3. Having clicked the link, the victim is requested to authenticate and authorize the client site to have access.

4. After completion of the authorization process, the authorization server redirects the user agent to the attacker's web site instead of the original client web site.

5. The attacker obtains the authorization "code" from his web site by means that are out of scope of this document.

6. He then constructs a redirect URI to the target web site (or application) based on the original authorization request's redirect URI and the newly obtained authorization "code", and directs his user agent to this URL. The authorization "code" is injected into the original client site (or application).

7. The client site uses the authorization "code" to fetch a token from the authorization server and associates this token with the attacker's user account on this site.

8. The attacker may now access the victim's resources using the client site.

Impact: The attacker gains access to the victim's resources as associated with his account on the client site.

Countermeasures:

- The attacker will need to use another redirect URI for its authorization process rather than the target web site because it needs to intercept the flow. So, if the authorization server associates the authorization "code" with the redirect URI of a particular end-user authorization and validates this redirect URI with the redirect URI passed to the token's endpoint, such an attack is detected (see Section 5.2.4.5).

- The authorization server may also enforce the usage and validation of pre-registered redirect URIs (see Section 5.2.3.5). This will allow for early recognition of authorization "code" disclosure to counterfeit clients.

- For native applications, one could also consider using deployment-specific client ids and secrets (see Section 5.2.3.4), along with the binding of authorization "codes" to "client_ids" (see Section 5.2.4.4) to detect such an attack because the attacker does not have access to the deployment-specific secret. Thus, he will not be able to exchange the authorization "code".

- o The client may consider using other flows that are not vulnerable to this kind of attack, such as the implicit grant type (see Section 4.4.2) or resource owner password credentials (see Section 4.4.3).

4.4.1.8. Threat: CSRF Attack against redirect-uri

Cross-site request forgery (CSRF) is a web-based attack whereby HTTP requests are transmitted from a user that the web site trusts or has authenticated (e.g., via HTTP redirects or HTML forms). CSRF attacks on OAuth approvals can allow an attacker to obtain authorization to OAuth protected resources without the consent of the user.

This attack works against the redirect URI used in the authorization "code" flow. An attacker could authorize an authorization "code" to their own protected resources on an authorization server. He then aborts the redirect flow back to the client on his device and tricks the victim into executing the redirect back to the client. The client receives the redirect, fetches the token(s) from the authorization server, and associates the victim's client session with the resources accessible using the token.

Impact: The user accesses resources on behalf of the attacker. The effective impact depends on the type of resource accessed. For example, the user may upload private items to an attacker's resources. Or, when using OAuth in 3rd-party login scenarios, the user may associate his client account with the attacker's identity at the external Identity Provider. In this way, the attacker could easily access the victim's data at the client by logging in from another device with his credentials at the external Identity Provider.

Countermeasures:

- o The "state" parameter should be used to link the authorization request with the redirect URI used to deliver the access token (Section 5.3.5).

- o Client developers and end users can be educated to not follow untrusted URLs.

4.4.1.9. Threat: Clickjacking Attack against Authorization

With clickjacking, a malicious site loads the target site in a transparent iFrame (see [iFrame]) overlaid on top of a set of dummy buttons that are carefully constructed to be placed directly under important buttons on the target site. When a user clicks a visible button, they are actually clicking a button (such as an "Authorize" button) on the hidden page.

Impact: An attacker can steal a user's authentication credentials and access their resources.

Countermeasures:

- For newer browsers, avoidance of iFrames during authorization can be enforced on the server side by using the X-FRAME-OPTIONS header (Section 5.2.2.6).

- For older browsers, JavaScript frame-busting (see [Framebusting]) techniques can be used but may not be effective in all browsers.

4.4.1.10. Threat: Resource Owner Impersonation

When a client requests access to protected resources, the authorization flow normally involves the resource owner's explicit response to the access request, either granting or denying access to the protected resources. A malicious client can exploit knowledge of the structure of this flow in order to gain authorization without the resource owner's consent, by transmitting the necessary requests programmatically and simulating the flow against the authorization server. That way, the client may gain access to the victim's resources without her approval. An authorization server will be vulnerable to this threat if it uses non-interactive authentication mechanisms or splits the authorization flow across multiple pages.

The malicious client might embed a hidden HTML user agent, interpret the HTML forms sent by the authorization server, and automatically send the corresponding form HTTP POST requests. As a prerequisite, the attacker must be able to execute the authorization process in the context of an already-authenticated session of the resource owner with the authorization server. There are different ways to achieve this:

- The malicious client could abuse an existing session in an external browser or cross-browser cookies on the particular device.

- The malicious client could also request authorization for an initial scope acceptable to the user and then silently abuse the resulting session in his browser instance to "silently" request another scope.

- Alternatively, the attacker might exploit an authorization server's ability to authenticate the resource owner automatically and without user interactions, e.g., based on certificates.

In all cases, such an attack is limited to clients running on the victim's device, either within the user agent or as a native app.

Please note: Such attacks cannot be prevented using CSRF countermeasures, since the attacker just "executes" the URLs as prepared by the authorization server including any nonce, etc.

Countermeasures:

Authorization servers should decide, based on an analysis of the risk associated with this threat, whether to detect and prevent this threat.

In order to prevent such an attack, the authorization server may force a user interaction based on non-predictable input values as part of the user consent approval. The authorization server could

- combine password authentication and user consent in a single form,

- make use of CAPTCHAs, or

- use one-time secrets sent out of band to the resource owner (e.g., via text or instant message).

Alternatively, in order to allow the resource owner to detect abuse, the authorization server could notify the resource owner of any approval by appropriate means, e.g., text or instant message, or email.

4.4.1.11. Threat: DoS Attacks That Exhaust Resources

If an authorization server includes a nontrivial amount of entropy in authorization "codes" or access tokens (limiting the number of possible codes/tokens) and automatically grants either without user intervention and has no limit on codes or access tokens per user, an attacker could exhaust the pool of authorization "codes" by repeatedly directing the user's browser to request authorization "codes" or access tokens.

Countermeasures:

- o The authorization server should consider limiting the number of access tokens granted per user.

- o The authorization server should include a nontrivial amount of entropy in authorization "codes".

4.4.1.12. Threat: DoS Using Manufactured Authorization "codes"

An attacker who owns a botnet can locate the redirect URIs of clients that listen on HTTP, access them with random authorization "codes", and cause a large number of HTTPS connections to be concentrated onto the authorization server. This can result in a denial-of-service (DoS) attack on the authorization server.

This attack can still be effective even when CSRF defense/the "state" parameter (see Section 4.4.1.8) is deployed on the client side. With such a defense, the attacker might need to incur an additional HTTP request to obtain a valid CSRF code/"state" parameter. This apparently cuts down the effectiveness of the attack by a factor of 2. However, if the HTTPS/HTTP cost ratio is higher than 2 (the cost factor is estimated to be around 3.5x at [SSL-Latency]), the attacker still achieves a magnification of resource utilization at the expense of the authorization server.

Impact: There are a few effects that the attacker can accomplish with this OAuth flow that they cannot easily achieve otherwise.

1. Connection laundering: With the clients as the relay between the attacker and the authorization server, the authorization server learns little or no information about the identity of the attacker. Defenses such as rate-limiting on the offending attacker machines are less effective because it is difficult to identify the attacking machines. Although an attacker could also launder its connections through an anonymizing system such as Tor, the effectiveness of that approach depends on the capacity of the anonymizing system. On the other hand, a potentially large number of OAuth clients could be utilized for this attack.

2. Asymmetric resource utilization: The attacker incurs the cost of an HTTP connection and causes an HTTPS connection to be made on the authorization server; the attacker can coordinate the timing of such HTTPS connections across multiple clients relatively easily. Although the attacker could achieve something similar, say, by including an iFrame pointing to the HTTPS URL of the authorization server in an HTTP web page and luring web users to visit that page, timing attacks using such a scheme may be more

difficult, as it seems nontrivial to synchronize a large number of users to simultaneously visit a particular site under the attacker's control.

Countermeasures:

- Though not a complete countermeasure by themselves, CSRF defense and the "state" parameter created with secure random codes should be deployed on the client side. The client should forward the authorization "code" to the authorization server only after both the CSRF token and the "state" parameter are validated.

- If the client authenticates the user, either through a single-sign-on protocol or through local authentication, the client should suspend the access by a user account if the number of invalid authorization "codes" submitted by this user exceeds a certain threshold.

- The authorization server should send an error response to the client reporting an invalid authorization "code" and rate-limit or disallow connections from clients whose number of invalid requests exceeds a threshold.

4.4.1.13. Threat: Code Substitution (OAuth Login)

An attacker could attempt to log into an application or web site using a victim's identity. Applications relying on identity data provided by an OAuth protected service API to login users are vulnerable to this threat. This pattern can be found in so-called "social login" scenarios.

As a prerequisite, a resource server offers an API to obtain personal information about a user that could be interpreted as having obtained a user identity. In this sense, the client is treating the resource server API as an "identity" API. A client utilizes OAuth to obtain an access token for the identity API. It then queries the identity API for an identifier and uses it to look up its internal user account data (login). The client assumes that, because it was able to obtain information about the user, the user has been authenticated.

If the client uses the grant type "code", the attacker needs to gather a valid authorization "code" of the respective victim from the same Identity Provider used by the target client application. The attacker tricks the victim into logging into a malicious app (which may appear to be legitimate to the Identity Provider) using the same Identity Provider as the target application. This results in the Identity Provider's authorization server issuing an authorization

"code" for the respective identity API. The malicious app then sends this code to the attacker, which in turn triggers a login process within the target application. The attacker now manipulates the authorization response and substitutes their code (bound to their identity) for the victim's code. This code is then exchanged by the client for an access token, which in turn is accepted by the identity API, since the audience, with respect to the resource server, is correct. But since the identifier returned by the identity API is determined by the identity in the access token (issued based on the victim's code), the attacker is logged into the target application under the victim's identity.

Impact: The attacker gains access to an application and user-specific data within the application.

Countermeasures:

- o All clients must indicate their client ids with every request to exchange an authorization "code" for an access token. The authorization server must validate whether the particular authorization "code" has been issued to the particular client. If possible, the client shall be authenticated beforehand.

- o Clients should use an appropriate protocol, such as OpenID (cf. [OPENID]) or SAML (cf. [OASIS.sstc-saml-bindings-1.1]) to implement user login. Both support audience restrictions on clients.

4.4.2. Implicit Grant

In the implicit grant type flow, the access token is directly returned to the client as a fragment part of the redirect URI. It is assumed that the token is not sent to the redirect URI target, as HTTP user agents do not send the fragment part of URIs to HTTP servers. Thus, an attacker cannot eavesdrop the access token on this communication path, and the token cannot leak through HTTP referrer headers.

4.4.2.1. Threat: Access Token Leak in Transport/Endpoints

This token might be eavesdropped by an attacker. The token is sent from the server to the client via a URI fragment of the redirect URI. If the communication is not secured or the endpoint is not secured, the token could be leaked by parsing the returned URI.

Impact: The attacker would be able to assume the same rights granted by the token.

Countermeasures:

- The authorization server should ensure confidentiality (e.g., using TLS) of the response from the authorization server to the client (see Section 5.1.1).

4.4.2.2. Threat: Access Token Leak in Browser History

An attacker could obtain the token from the browser's history. Note that this means the attacker needs access to the particular device.

Countermeasures:

- Use short expiry time for tokens (see Section 5.1.5.3). Reduced scope of the token may reduce the impact of that attack (see Section 5.1.5.1).

- Make responses non-cacheable.

4.4.2.3. Threat: Malicious Client Obtains Authorization

A malicious client could attempt to obtain a token by fraud.

The same countermeasures as for Section 4.4.1.4 are applicable, except client authentication.

4.4.2.4. Threat: Manipulation of Scripts

A hostile party could act as the client web server and replace or modify the actual implementation of the client (script). This could be achieved using DNS or ARP spoofing. This applies to clients implemented within the web browser in a scripting language.

Impact: The attacker could obtain user credential information and assume the full identity of the user.

Countermeasures:

- The authorization server should authenticate the server from which scripts are obtained (see Section 5.1.2).

- The client should ensure that scripts obtained have not been altered in transport (see Section 5.1.1).

- o Introduce one-time, per-use secrets (e.g., "client_secret") values that can only be used by scripts in a small time window once loaded from a server. The intention would be to reduce the effectiveness of copying client-side scripts for re-use in an attacker's modified code.

4.4.2.5. Threat: CSRF Attack against redirect-uri

CSRF attacks (see Section 4.4.1.8) also work against the redirect URI used in the implicit grant flow. An attacker could acquire an access token to their own protected resources. He could then construct a redirect URI and embed their access token in that URI. If he can trick the user into following the redirect URI and the client does not have protection against this attack, the user may have the attacker's access token authorized within their client.

Impact: The user accesses resources on behalf of the attacker. The effective impact depends on the type of resource accessed. For example, the user may upload private items to an attacker's resources. Or, when using OAuth in 3rd-party login scenarios, the user may associate his client account with the attacker's identity at the external Identity Provider. In this way, the attacker could easily access the victim's data at the client by logging in from another device with his credentials at the external Identity Provider.

Countermeasures:

- o The "state" parameter should be used to link the authorization request with the redirect URI used to deliver the access token. This will ensure that the client is not tricked into completing any redirect callback unless it is linked to an authorization request initiated by the client. The "state" parameter should not be guessable, and the client should be capable of keeping the "state" parameter secret.

- o Client developers and end users can be educated to not follow untrusted URLs.

4.4.2.6. Threat: Token Substitution (OAuth Login)

An attacker could attempt to log into an application or web site using a victim's identity. Applications relying on identity data provided by an OAuth protected service API to login users are vulnerable to this threat. This pattern can be found in so-called "social login" scenarios.

As a prerequisite, a resource server offers an API to obtain personal information about a user that could be interpreted as having obtained a user identity. In this sense, the client is treating the resource server API as an "identity" API. A client utilizes OAuth to obtain an access token for the identity API. It then queries the identity API for an identifier and uses it to look up its internal user account data (login). The client assumes that, because it was able to obtain information about the user, the user has been authenticated.

To succeed, the attacker needs to gather a valid access token of the respective victim from the same Identity Provider used by the target client application. The attacker tricks the victim into logging into a malicious app (which may appear to be legitimate to the Identity Provider) using the same Identity Provider as the target application. This results in the Identity Provider's authorization server issuing an access token for the respective identity API. The malicious app then sends this access token to the attacker, which in turn triggers a login process within the target application. The attacker now manipulates the authorization response and substitutes their access token (bound to their identity) for the victim's access token. This token is accepted by the identity API, since the audience, with respect to the resource server, is correct. But since the identifier returned by the identity API is determined by the identity in the access token, the attacker is logged into the target application under the victim's identity.

Impact: The attacker gains access to an application and user-specific data within the application.

Countermeasures:

- o Clients should use an appropriate protocol, such as OpenID (cf. [OPENID]) or SAML (cf. [OASIS.sstc-saml-bindings-1.1]) to implement user login. Both support audience restrictions on clients.

4.4.3. Resource Owner Password Credentials

The resource owner password credentials grant type (see [RFC6749], Section 4.3), often used for legacy/migration reasons, allows a client to request an access token using an end-user's user id and password along with its own credential. This grant type has higher risk because it maintains the UID/password anti-pattern. Additionally, because the user does not have control over the authorization process, clients using this grant type are not limited

by scope but instead have potentially the same capabilities as the user themselves. As there is no authorization step, the ability to offer token revocation is bypassed.

Because passwords are often used for more than 1 service, this anti-pattern may also put at risk whatever else is accessible with the supplied credential. Additionally, any easily derived equivalent (e.g., joe@example.com and joe@example.net) might easily allow someone to guess that the same password can be used elsewhere.

Impact: The resource server can only differentiate scope based on the access token being associated with a particular client. The client could also acquire long-lived tokens and pass them up to an attacker's web service for further abuse. The client, eavesdroppers, or endpoints could eavesdrop the user id and password.

Countermeasures:

- Except for migration reasons, minimize use of this grant type.
- The authorization server should validate the client id associated with the particular refresh token with every refresh request (Section 5.2.2.2).
- As per the core OAuth specification, the authorization server must ensure that these transmissions are protected using transport-layer mechanisms such as TLS (see Section 5.1.1).
- Rather than encouraging users to use a UID and password, service providers should instead encourage users not to use the same password for multiple services.
- Limit use of resource owner password credential grants to scenarios where the client application and the authorizing service are from the same organization.

4.4.3.1. Threat: Accidental Exposure of Passwords at Client Site

If the client does not provide enough protection, an attacker or disgruntled employee could retrieve the passwords for a user.

Countermeasures:

- Use other flows that do not rely on the client's cooperation for secure resource owner credential handling.
- Use digest authentication instead of plaintext credential processing.

- o Obfuscate passwords in logs.

4.4.3.2. Threat: Client Obtains Scopes without End-User Authorization

All interaction with the resource owner is performed by the client. Thus it might, intentionally or unintentionally, happen that the client obtains a token with scope unknown for, or unintended by, the resource owner. For example, the resource owner might think the client needs and acquires read-only access to its media storage only but the client tries to acquire an access token with full access permissions.

Countermeasures:

- o Use other flows that do not rely on the client's cooperation for resource owner interaction.

- o The authorization server may generally restrict the scope of access tokens (Section 5.1.5.1) issued by this flow. If the particular client is trustworthy and can be authenticated in a reliable way, the authorization server could relax that restriction. Resource owners may prescribe (e.g., in their preferences) what the maximum scope is for clients using this flow.

- o The authorization server could notify the resource owner by an appropriate medium, e.g., email, of the grant issued (see Section 5.1.3).

4.4.3.3. Threat: Client Obtains Refresh Token through Automatic Authorization

All interaction with the resource owner is performed by the client. Thus it might, intentionally or unintentionally, happen that the client obtains a long-term authorization represented by a refresh token even if the resource owner did not intend so.

Countermeasures:

- o Use other flows that do not rely on the client's cooperation for resource owner interaction.

- o The authorization server may generally refuse to issue refresh tokens in this flow (see Section 5.2.2.1). If the particular client is trustworthy and can be authenticated in a reliable way (see client authentication), the authorization server could relax

that restriction. Resource owners may allow or deny (e.g., in their preferences) the issuing of refresh tokens using this flow as well.

- The authorization server could notify the resource owner by an appropriate medium, e.g., email, of the refresh token issued (see Section 5.1.3).

4.4.3.4. Threat: Obtaining User Passwords on Transport

An attacker could attempt to eavesdrop the transmission of end-user credentials with the grant type "password" between the client and server.

Impact: Disclosure of a single end-user's password.

Countermeasures:

- Ensure confidentiality of requests (Section 5.1.1).
- Use alternative authentication means that do not require the sending of plaintext credentials over the wire (e.g., Hash-based Message Authentication Code).

4.4.3.5. Threat: Obtaining User Passwords from Authorization Server Database

An attacker may obtain valid username/password combinations from the authorization server's database by gaining access to the database or launching a SQL injection attack.

Impact: Disclosure of all username/password combinations. The impact may exceed the domain of the authorization server, since many users tend to use the same credentials on different services.

Countermeasures:

- Enforce credential storage protection best practices (Section 5.1.4.1).

4.4.3.6. Threat: Online Guessing

An attacker may try to guess valid username/password combinations using the grant type "password".

Impact: Revelation of a single username/password combination.

Countermeasures:

- o Utilize secure password policy (Section 5.1.4.2.1).
- o Lock accounts (Section 5.1.4.2.3).
- o Use tar pit (Section 5.1.4.2.4).
- o Use CAPTCHAs (Section 5.1.4.2.5).
- o Consider not using the grant type "password".
- o Client authentication (see Section 5.2.3) will provide another authentication factor and thus hinder the attack.

4.4.4. Client Credentials

Client credentials (see [RFC6749], Section 3) consist of an identifier (not secret) combined with an additional means (such as a matching client secret) of authenticating a client. The threats to this grant type are similar to those described in Section 4.4.3.

4.5. Refreshing an Access Token

4.5.1. Threat: Eavesdropping Refresh Tokens from Authorization Server

An attacker may eavesdrop refresh tokens when they are transmitted from the authorization server to the client.

Countermeasures:

- o As per the core OAuth spec, the authorization servers must ensure that these transmissions are protected using transport-layer mechanisms such as TLS (see Section 5.1.1).
- o If end-to-end confidentiality cannot be guaranteed, reducing scope (see Section 5.1.5.1) and expiry time (see Section 5.1.5.3) for issued access tokens can be used to reduce the damage in case of leaks.

4.5.2. Threat: Obtaining Refresh Token from Authorization Server Database

This threat is applicable if the authorization server stores refresh tokens as handles in a database. An attacker may obtain refresh tokens from the authorization server's database by gaining access to the database or launching a SQL injection attack.

Impact: Disclosure of all refresh tokens.

Countermeasures:

- o Enforce credential storage protection best practices (Section 5.1.4.1).
- o Bind token to client id, if the attacker cannot obtain the required id and secret (Section 5.1.5.8).

4.5.3. Threat: Obtaining Refresh Token by Online Guessing

An attacker may try to guess valid refresh token values and send it using the grant type "refresh_token" in order to obtain a valid access token.

Impact: Exposure of a single refresh token and derivable access tokens.

Countermeasures:

- o For handle-based designs (Section 5.1.4.2.2).
- o For assertion-based designs (Section 5.1.5.9).
- o Bind token to client id, because the attacker would guess the matching client id, too (see Section 5.1.5.8).
- o Authenticate the client; this adds another element that the attacker has to guess (see Section 5.2.3.4).

4.5.4. Threat: Refresh Token Phishing by Counterfeit Authorization Server

An attacker could try to obtain valid refresh tokens by proxying requests to the authorization server. Given the assumption that the authorization server URL is well-known at development time or can at least be obtained from a well-known resource server, the attacker must utilize some kind of spoofing in order to succeed.

Countermeasures:

- o Utilize server authentication (as described in Section 5.1.2).

4.6. Accessing Protected Resources

4.6.1. Threat: Eavesdropping Access Tokens on Transport

An attacker could try to obtain a valid access token on transport between the client and resource server. As access tokens are shared secrets between the authorization server and resource server, they should be treated with the same care as other credentials (e.g., end-user passwords).

Countermeasures:

- Access tokens sent as bearer tokens should not be sent in the clear over an insecure channel. As per the core OAuth spec, transmission of access tokens must be protected using transport-layer mechanisms such as TLS (see Section 5.1.1).

- A short lifetime reduces impact in case tokens are compromised (see Section 5.1.5.3).

- The access token can be bound to a client's identifier and require the client to prove legitimate ownership of the token to the resource server (see Section 5.4.2).

4.6.2. Threat: Replay of Authorized Resource Server Requests

An attacker could attempt to replay valid requests in order to obtain or to modify/destroy user data.

Countermeasures:

- The resource server should utilize transport security measures (e.g., TLS) in order to prevent such attacks (see Section 5.1.1). This would prevent the attacker from capturing valid requests.

- Alternatively, the resource server could employ signed requests (see Section 5.4.3) along with nonces and timestamps in order to uniquely identify requests. The resource server should detect and refuse every replayed request.

4.6.3. Threat: Guessing Access Tokens

Where the token is a handle, the attacker may attempt to guess the access token values based on knowledge they have from other access tokens.

Impact: Access to a single user's data.

Countermeasures:

- Handle tokens should have a reasonable level of entropy (see Section 5.1.4.2.2) in order to make guessing a valid token value infeasible.

- Assertion (or self-contained token) token contents should be protected by a digital signature (see Section 5.1.5.9).

- Security can be further strengthened by using a short access token duration (see Sections 5.1.5.2 and 5.1.5.3).

4.6.4. Threat: 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.

Countermeasures:

- Clients should not make authenticated requests with an access token to unfamiliar resource servers, regardless of the presence of a secure channel. If the resource server URL is well-known to the client, it may authenticate the resource servers (see Section 5.1.2).

- Associate the endpoint URL of the resource server the client talked to with the access token (e.g., in an audience field) and validate the association at a legitimate resource server. The endpoint URL validation policy may be strict (exact match) or more relaxed (e.g., same host). This would require telling the authorization server about the resource server endpoint URL in the authorization process.

- Associate an access token with a client and authenticate the client with resource server requests (typically via a signature, in order to not disclose a secret to a potential attacker). This prevents the attack because the counterfeit server is assumed to lack the capability to correctly authenticate on behalf of the legitimate client to the resource server (Section 5.4.2).

- Restrict the token scope (see Section 5.1.5.1) and/or limit the token to a certain resource server (Section 5.1.5.5).

4.6.5. Threat: Abuse of Token by Legitimate Resource Server or Client

A legitimate resource server could attempt to use an access token to access another resource server. Similarly, a client could try to use a token obtained for one server on another resource server.

Countermeasures:

- Tokens should be restricted to particular resource servers (see Section 5.1.5.5).

4.6.6. Threat: Leak of Confidential Data in HTTP Proxies

An OAuth HTTP authentication scheme as discussed in [RFC6749] is optional. However, [RFC2616] relies on the Authorization and WWW-Authenticate headers to distinguish authenticated content so that it can be protected. Proxies and caches, in particular, may fail to adequately protect requests not using these headers. For example, private authenticated content may be stored in (and thus be retrievable from) publicly accessible caches.

Countermeasures:

- Clients and resource servers not using an OAuth HTTP authentication scheme (see Section 5.4.1) should take care to use Cache-Control headers to minimize the risk that authenticated content is not protected. Such clients should send a Cache-Control header containing the "no-store" option [RFC2616]. Resource server success (2XX status) responses to these requests should contain a Cache-Control header with the "private" option [RFC2616].

- Reducing scope (see Section 5.1.5.1) and expiry time (Section 5.1.5.3) for access tokens can be used to reduce the damage in case of leaks.

4.6.7. Threat: Token Leakage via Log Files and HTTP Referrers

If access tokens are sent via URI query parameters, such tokens may leak to log files and the HTTP "referer".

Countermeasures:

- Use Authorization headers or POST parameters instead of URI request parameters (see Section 5.4.1).

- Set logging configuration appropriately.

- o Prevent unauthorized persons from access to system log files (see Section 5.1.4.1.1).

- o Abuse of leaked access tokens can be prevented by enforcing authenticated requests (see Section 5.4.2).

- o The impact of token leakage may be reduced by limiting scope (see Section 5.1.5.1) and duration (see Section 5.1.5.3) and by enforcing one-time token usage (see Section 5.1.5.4).

5. Security Considerations

This section describes the countermeasures as recommended to mitigate the threats described in Section 4.

5.1. General

This section covers considerations that apply generally across all OAuth components (client, resource server, token server, and user agents).

5.1.1. Ensure Confidentiality of Requests

This is applicable to all requests sent from the client to the authorization server or resource server. While OAuth provides a mechanism for verifying the integrity of requests, it provides no guarantee of request confidentiality. Unless further precautions are taken, eavesdroppers will have full access to request content and may be able to mount interception or replay attacks by using the contents of requests, e.g., secrets or tokens.

Attacks can be mitigated by using transport-layer mechanisms such as TLS [RFC5246]. A virtual private network (VPN), e.g., based on IPsec VPNs [RFC4301], may be considered as well.

Note: This document assumes end-to-end TLS protected connections between the respective protocol entities. Deployments deviating from this assumption by offloading TLS in between (e.g., on the data center edge) must refine this threat model in order to account for the additional (mainly insider) threat this may cause.

This is a countermeasure against the following threats:

- o Replay of access tokens obtained on the token's endpoint or the resource server's endpoint

- o Replay of refresh tokens obtained on the token's endpoint

- o Replay of authorization "codes" obtained on the token's endpoint (redirect?)

- o Replay of user passwords and client secrets

5.1.2. Utilize Server Authentication

HTTPS server authentication or similar means can be used to authenticate the identity of a server. The goal is to reliably bind the fully qualified domain name of the server to the public key presented by the server during connection establishment (see [RFC2818]).

The client should validate the binding of the server to its domain name. If the server fails to prove that binding, the communication is considered a man-in-the-middle attack. This security measure depends on the certification authorities the client trusts for that purpose. Clients should carefully select those trusted CAs and protect the storage for trusted CA certificates from modifications.

This is a countermeasure against the following threats:

- o Spoofing

- o Proxying

- o Phishing by counterfeit servers

5.1.3. Always Keep the Resource Owner Informed

Transparency to the resource owner is a key element of the OAuth protocol. The user should always be in control of the authorization processes and get the necessary information to make informed decisions. Moreover, user involvement is a further security countermeasure. The user can probably recognize certain kinds of attacks better than the authorization server. Information can be presented/exchanged during the authorization process, after the authorization process, and every time the user wishes to get informed by using techniques such as:

- o User consent forms.

- o Notification messages (e.g., email, SMS, ...). Note that notifications can be a phishing vector. Messages should be such that look-alike phishing messages cannot be derived from them.

- o Activity/event logs.

- o User self-care applications or portals.

5.1.4. Credentials

This section describes countermeasures used to protect all kinds of credentials from unauthorized access and abuse. Credentials are long-term secrets, such as client secrets and user passwords as well as all kinds of tokens (refresh and access tokens) or authorization "codes".

5.1.4.1. Enforce Credential Storage Protection Best Practices

Administrators should undertake industry best practices to protect the storage of credentials (for example, see [OWASP]). Such practices may include but are not limited to the following sub-sections.

5.1.4.1.1. Enforce Standard System Security Means

A server system may be locked down so that no attacker may get access to sensitive configuration files and databases.

5.1.4.1.2. Enforce Standard SQL Injection Countermeasures

If a client identifier or other authentication component is queried or compared against a SQL database, it may become possible for an injection attack to occur if parameters received are not validated before submission to the database.

- o Ensure that server code is using the minimum database privileges possible to reduce the "surface" of possible attacks.

- o Avoid dynamic SQL using concatenated input. If possible, use static SQL.

- o When using dynamic SQL, parameterize queries using bind arguments. Bind arguments eliminate the possibility of SQL injections.

- o Filter and sanitize the input. For example, if an identifier has a known format, ensure that the supplied value matches the identifier syntax rules.

5.1.4.1.3. No Cleartext Storage of Credentials

The authorization server should not store credentials in clear text. Typical approaches are to store hashes instead or to encrypt credentials. If the credential lacks a reasonable entropy level (because it is a user password), an additional salt will harden the storage to make offline dictionary attacks more difficult.

Note: Some authentication protocols require the authorization server to have access to the secret in the clear. Those protocols cannot be implemented if the server only has access to hashes. Credentials should be strongly encrypted in those cases.

5.1.4.1.4. Encryption of Credentials

For client applications, insecurely persisted client credentials are easy targets for attackers to obtain. Store client credentials using an encrypted persistence mechanism such as a keystore or database. Note that compiling client credentials directly into client code makes client applications vulnerable to scanning as well as difficult to administer should client credentials change over time.

5.1.4.1.5. Use of Asymmetric Cryptography

Usage of asymmetric cryptography will free the authorization server of the obligation to manage credentials.

5.1.4.2. Online Attacks on Secrets

5.1.4.2.1. Utilize Secure Password Policy

The authorization server may decide to enforce a complex user password policy in order to increase the user passwords' entropy to hinder online password attacks. Note that too much complexity can increase the likelihood that users re-use passwords or write them down, or otherwise store them insecurely.

5.1.4.2.2. Use High Entropy for Secrets

When creating secrets not intended for usage by human users (e.g., client secrets or token handles), the authorization server should include a reasonable level of entropy in order to mitigate the risk of guessing attacks. The token value should be >=128 bits long and constructed from a cryptographically strong random or pseudo-random number sequence (see [RFC4086] for best current practice) generated by the authorization server.

5.1.4.2.3. Lock Accounts

Online attacks on passwords can be mitigated by locking the respective accounts after a certain number of failed attempts.

Note: This measure can be abused to lock down legitimate service users.

5.1.4.2.4. Use Tar Pit

The authorization server may react on failed attempts to authenticate by username/password by temporarily locking the respective account and delaying the response for a certain duration. This duration may increase with the number of failed attempts. The objective is to slow the attacker's attempts on a certain username down.

Note: This may require a more complex and stateful design of the authorization server.

5.1.4.2.5. Use CAPTCHAs

The idea is to prevent programs from automatically checking a huge number of passwords, by requiring human interaction.

Note: This has a negative impact on user experience.

5.1.5. Tokens (Access, Refresh, Code)

5.1.5.1. Limit Token Scope

The authorization server may decide to reduce or limit the scope associated with a token. The basis of this decision is out of scope; examples are:

- a client-specific policy, e.g., issue only less powerful tokens to public clients,
- a service-specific policy, e.g., it is a very sensitive service,
- a resource-owner-specific setting, or
- combinations of such policies and preferences.

The authorization server may allow different scopes dependent on the grant type. For example, end-user authorization via direct interaction with the end user (authorization "code") might be considered more reliable than direct authorization via grant type "username"/"password". This means will reduce the impact of the following threats:

- token leakage
- token issuance to malicious software
- unintended issuance of powerful tokens with resource owner credentials flow

5.1.5.2. Determine Expiration Time

Tokens should generally expire after a reasonable duration. This complements and strengthens other security measures (such as signatures) and reduces the impact of all kinds of token leaks. Depending on the risk associated with token leakage, tokens may expire after a few minutes (e.g., for payment transactions) or stay valid for hours (e.g., read access to contacts).

The expiration time is determined by several factors, including:

- risk associated with token leakage,
- duration of the underlying access grant,
- duration until the modification of an access grant should take effect, and
- time required for an attacker to guess or produce a valid token.

5.1.5.3. Use Short Expiration Time

A short expiration time for tokens is a means of protection against the following threats:

- replay
- token leak (a short expiration time will reduce impact)
- online guessing (a short expiration time will reduce the likelihood of success)

Note: Short token duration requires more precise clock synchronization between the authorization server and resource server. Furthermore, shorter duration may require more token refreshes (access token) or repeated end-user authorization processes (authorization "code" and refresh token).

5.1.5.4. Limit Number of Usages or One-Time Usage

The authorization server may restrict the number of requests or operations that can be performed with a certain token. This mechanism can be used to mitigate the following threats:

- replay of tokens
- guessing

For example, if an authorization server observes more than one attempt to redeem an authorization "code", the authorization server may want to revoke all access tokens granted based on the authorization "code" as well as reject the current request.

As with the authorization "code", access tokens may also have a limited number of operations. This either forces client applications to re-authenticate and use a refresh token to obtain a fresh access token, or forces the client to re-authorize the access token by involving the user.

5.1.5.5. Bind Tokens to a Particular Resource Server (Audience)

Authorization servers in multi-service environments may consider issuing tokens with different content to different resource servers and to explicitly indicate in the token the target server to which a token is intended to be sent. SAML assertions (see [OASIS.saml-core-2.0-os]) use the Audience element for this purpose. This countermeasure can be used in the following situations:

- It reduces the impact of a successful replay attempt, since the token is applicable to a single resource server only.
- It prevents abuse of a token by a rogue resource server or client, since the token can only be used on that server. It is rejected by other servers.
- It reduces the impact of leakage of a valid token to a counterfeit resource server.

5.1.5.6. Use Endpoint Address as Token Audience

This may be used to indicate to a resource server which endpoint URL has been used to obtain the token. This measure will allow the detection of requests from a counterfeit resource server, since such a token will contain the endpoint URL of that server.

5.1.5.7. Use Explicitly Defined Scopes for Audience and Tokens

Deployments may consider only using tokens with explicitly defined scopes, where every scope is associated with a particular resource server. This approach can be used to mitigate attacks where a resource server or client uses a token for a different purpose than the one intended.

5.1.5.8. Bind Token to Client id

An authorization server may bind a token to a certain client identifier. This identifier should be validated for every request with that token. This technique can be used to

- o detect token leakage and
- o prevent token abuse.

Note: Validating the client identifier may require the target server to authenticate the client's identifier. This authentication can be based on secrets managed independently of the token (e.g., pre-registered client id/secret on authorization server) or sent with the token itself (e.g., as part of the encrypted token content).

5.1.5.9. Sign Self-Contained Tokens

Self-contained tokens should be signed in order to detect any attempt to modify or produce faked tokens (e.g., Hash-based Message Authentication Code or digital signatures).

5.1.5.10. Encrypt Token Content

Self-contained tokens may be encrypted for confidentiality reasons or to protect system internal data. Depending on token format, keys (e.g., symmetric keys) may have to be distributed between server nodes. The method of distribution should be defined by the token and the encryption used.

5.1.5.11. Adopt a Standard Assertion Format

For service providers intending to implement an assertion-based token design, it is highly recommended to adopt a standard assertion format (such as SAML [OASIS.saml-core-2.0-os] or the JavaScript Object Notation Web Token (JWT) [OAuth-JWT]).

5.1.6. Access Tokens

The following measures should be used to protect access tokens:

- o Keep them in transient memory (accessible by the client application only).
- o Pass tokens securely using secure transport (TLS).
- o Ensure that client applications do not share tokens with 3rd parties.

5.2. Authorization Server

This section describes considerations related to the OAuth authorization server endpoint.

5.2.1. Authorization "codes"

5.2.1.1. Automatic Revocation of Derived Tokens If Abuse Is Detected

If an authorization server observes multiple attempts to redeem an authorization grant (e.g., such as an authorization "code"), the authorization server may want to revoke all tokens granted based on the authorization grant.

5.2.2. Refresh Tokens

5.2.2.1. Restricted Issuance of Refresh Tokens

The authorization server may decide, based on an appropriate policy, not to issue refresh tokens. Since refresh tokens are long-term credentials, they may be subject to theft. For example, if the authorization server does not trust a client to securely store such tokens, it may refuse to issue such a client a refresh token.

5.2.2.2. Binding of Refresh Token to "client_id"

The authorization server should match every refresh token to the identifier of the client to whom it was issued. The authorization server should check that the same "client_id" is present for every request to refresh the access token. If possible (e.g., confidential clients), the authorization server should authenticate the respective client.

This is a countermeasure against refresh token theft or leakage.

Note: This binding should be protected from unauthorized modifications.

5.2.2.3. Refresh Token Rotation

Refresh token rotation is intended to automatically detect and prevent attempts to use the same refresh token in parallel from different apps/devices. This happens if a token gets stolen from the client and is subsequently used by both the attacker and the legitimate client. The basic idea is to change the refresh token value with every refresh request in order to detect attempts to obtain access tokens using old refresh tokens. Since the authorization server cannot determine whether the attacker or the legitimate client is trying to access, in case of such an access attempt the valid refresh token and the access authorization associated with it are both revoked.

The OAuth specification supports this measure in that the token's response allows the authorization server to return a new refresh token even for requests with grant type "refresh_token".

Note: This measure may cause problems in clustered environments, since usage of the currently valid refresh token must be ensured. In such an environment, other measures might be more appropriate.

5.2.2.4. Revocation of Refresh Tokens

The authorization server may allow clients or end users to explicitly request the invalidation of refresh tokens. A mechanism to revoke tokens is specified in [OAuth-REVOCATION].

This is a countermeasure against:

- o device theft,
- o impersonation of a resource owner, or
- o suspected compromised client applications.

5.2.2.5. Device Identification

The authorization server may require the binding of authentication credentials to a device identifier. The International Mobile Station Equipment Identity [IMEI] is one example of such an identifier; there are also operating system-specific identifiers. The authorization server could include such an identifier when authenticating user credentials in order to detect token theft from a particular device.

Note: Any implementation should consider potential privacy implications of using device identifiers.

5.2.2.6. X-FRAME-OPTIONS Header

For newer browsers, avoidance of iFrames can be enforced on the server side by using the X-FRAME-OPTIONS header (see [X-Frame-Options]). This header can have two values, "DENY" and "SAMEORIGIN", which will block any framing or any framing by sites with a different origin, respectively. The value "ALLOW-FROM" specifies a list of trusted origins that iFrames may originate from.

This is a countermeasure against the following threat:

- o Clickjacking attacks

5.2.3. Client Authentication and Authorization

As described in Section 3 (Security Features), clients are identified, authenticated, and authorized for several purposes, such as to:

- o Collate requests to the same client,
- o Indicate to the user that the client is recognized by the authorization server,
- o Authorize access of clients to certain features on the authorization server or resource server, and
- o Log a client identifier to log files for analysis or statistics.

Due to the different capabilities and characteristics of the different client types, there are different ways to support these objectives, which will be described in this section. Authorization server providers should be aware of the security policy and deployment of a particular client and adapt its treatment accordingly. For example, one approach could be to treat all clients as less trustworthy and unsecure. On the other extreme, a service provider could activate every client installation individually by an administrator and in that way gain confidence in the identity of the software package and the security of the environment in which the client is installed. There are several approaches in between.

5.2.3.1. Don't Issue Secrets to Clients with Inappropriate Security Policy

Authorization servers should not issue secrets to clients that cannot protect secrets ("public" clients). This reduces the probability of the server treating the client as strongly authenticated.

For example, it is of limited benefit to create a single client id and secret that are shared by all installations of a native application. Such a scenario requires that this secret must be transmitted from the developer via the respective distribution channel, e.g., an application market, to all installations of the application on end-user devices. A secret, burned into the source code of the application or an associated resource bundle, is not protected from reverse engineering. Secondly, such secrets cannot be revoked, since this would immediately put all installations out of work. Moreover, since the authorization server cannot really trust the client's identifier, it would be dangerous to indicate to end users the trustworthiness of the client.

There are other ways to achieve a reasonable security level, as described in the following sections.

5.2.3.2. Require User Consent for Public Clients without Secret

Authorization servers should not allow automatic authorization for public clients. The authorization server may issue an individual client id but should require that all authorizations are approved by the end user. For clients without secrets, this is a countermeasure against the following threat:

- Impersonation of public client applications.

5.2.3.3. Issue a "client_id" Only in Combination with "redirect_uri"

The authorization server may issue a "client_id" and bind the "client_id" to a certain pre-configured "redirect_uri". Any authorization request with another redirect URI is refused automatically. Alternatively, the authorization server should not accept any dynamic redirect URI for such a "client_id" and instead should always redirect to the well-known pre-configured redirect URI. This is a countermeasure for clients without secrets against the following threats:

- Cross-site scripting attacks

- Impersonation of public client applications

5.2.3.4. Issue Installation-Specific Client Secrets

An authorization server may issue separate client identifiers and corresponding secrets to the different installations of a particular client (i.e., software package). The effect of such an approach would be to turn otherwise "public" clients back into "confidential" clients.

For web applications, this could mean creating one "client_id" and "client_secret" for each web site on which a software package is installed. So, the provider of that particular site could request a client id and secret from the authorization server during the setup of the web site. This would also allow the validation of some of the properties of that web site, such as redirect URI, web site URL, and whatever else proves useful. The web site provider has to ensure the security of the client secret on the site.

For native applications, things are more complicated because every copy of a particular application on any device is a different installation. Installation-specific secrets in this scenario will require obtaining a "client_id" and "client_secret" either

1. during the download process from the application market, or

2. during installation on the device.

Either approach will require an automated mechanism for issuing client ids and secrets, which is currently not defined by OAuth.

The first approach would allow the achievement of a certain level of trust in the authenticity of the application, whereas the second option only allows the authentication of the installation but not the validation of properties of the client. But this would at least help

to prevent several replay attacks. Moreover, installation-specific "client_ids" and secrets allow the selective revocation of all refresh tokens of a specific installation at once.

5.2.3.5. Validate Pre-Registered "redirect_uri"

An authorization server should require all clients to register their "redirect_uri", and the "redirect_uri" should be the full URI as defined in [RFC6749]. The way that this registration is performed is out of scope of this document. As per the core spec, every actual redirect URI sent with the respective "client_id" to the end-user authorization endpoint must match the registered redirect URI. Where it does not match, the authorization server should assume that the inbound GET request has been sent by an attacker and refuse it. Note: The authorization server should not redirect the user agent back to the redirect URI of such an authorization request. Validating the pre-registered "redirect_uri" is a countermeasure against the following threats:

- Authorization "code" leakage through counterfeit web site: allows authorization servers to detect attack attempts after the first redirect to an end-user authorization endpoint (Section 4.4.1.7).
- Open redirector attack via a client redirection endpoint (Section 4.1.5).
- Open redirector phishing attack via an authorization server redirection endpoint (Section 4.2.4).

The underlying assumption of this measure is that an attacker will need to use another redirect URI in order to get access to the authorization "code". Deployments might consider the possibility of an attacker using spoofing attacks to a victim's device to circumvent this security measure.

Note: Pre-registering clients might not scale in some deployments (manual process) or require dynamic client registration (not specified yet). With the lack of dynamic client registration, a pre-registered "redirect_uri" only works for clients bound to certain deployments at development/configuration time. As soon as dynamic resource server discovery is required, the pre-registered "redirect_uri" may no longer be feasible.

5.2.3.6. Revoke Client Secrets

An authorization server may revoke a client's secret in order to prevent abuse of a revealed secret.

Note: This measure will immediately invalidate any authorization "code" or refresh token issued to the respective client. This might unintentionally impact client identifiers and secrets used across multiple deployments of a particular native or web application.

This a countermeasure against:

- Abuse of revealed client secrets for private clients

5.2.3.7. Use Strong Client Authentication (e.g., client_assertion/ client_token)

By using an alternative form of authentication such as client assertion [OAuth-ASSERTIONS], the need to distribute a "client_secret" is eliminated. This may require the use of a secure private key store or other supplemental authentication system as specified by the client assertion issuer in its authentication process.

5.2.4. End-User Authorization

This section includes considerations for authorization flows involving the end user.

5.2.4.1. Automatic Processing of Repeated Authorizations Requires Client Validation

Authorization servers should NOT automatically process repeat authorizations where the client is not authenticated through a client secret or some other authentication mechanism such as a signed authentication assertion certificate (Section 5.2.3.7) or validation of a pre-registered redirect URI (Section 5.2.3.5).

5.2.4.2. Informed Decisions Based on Transparency

The authorization server should clearly explain to the end user what happens in the authorization process and what the consequences are. For example, the user should understand what access he is about to grant to which client for what duration. It should also be obvious to the user whether the server is able to reliably certify certain client properties (web site URL, security policy).

5.2.4.3. Validation of Client Properties by End User

In the authorization process, the user is typically asked to approve a client's request for authorization. This is an important security mechanism by itself because the end user can be involved in the validation of client properties, such as whether the client name known to the authorization server fits the name of the web site or the application the end user is using. This measure is especially helpful in situations where the authorization server is unable to authenticate the client. It is a countermeasure against:

- A malicious application

- A client application masquerading as another client

5.2.4.4. Binding of Authorization "code" to "client_id"

The authorization server should bind every authorization "code" to the id of the respective client that initiated the end-user authorization process. This measure is a countermeasure against:

- Replay of authorization "codes" with different client credentials, since an attacker cannot use another "client_id" to exchange an authorization "code" into a token

- Online guessing of authorization "codes"

Note: This binding should be protected from unauthorized modifications (e.g., using protected memory and/or a secure database).

5.2.4.5. Binding of Authorization "code" to "redirect_uri"

The authorization server should be able to bind every authorization "code" to the actual redirect URI used as the redirect target of the client in the end-user authorization process. This binding should be validated when the client attempts to exchange the respective authorization "code" for an access token. This measure is a countermeasure against authorization "code" leakage through counterfeit web sites, since an attacker cannot use another redirect URI to exchange an authorization "code" into a token.

5.3. Client App Security

This section deals with considerations for client applications.

5.3.1. Don't Store Credentials in Code or Resources Bundled with Software Packages

Because of the number of copies of client software, there is limited benefit in creating a single client id and secret that is shared by all installations of an application. Such an application by itself would be considered a "public" client, as it cannot be presumed to be able to keep client secrets. A secret, burned into the source code of the application or an associated resource bundle, cannot be protected from reverse engineering. Secondly, such secrets cannot be revoked, since this would immediately put all installations out of work. Moreover, since the authorization server cannot really trust the client's identifier, it would be dangerous to indicate to end users the trustworthiness of the client.

5.3.2. Use Standard Web Server Protection Measures (for Config Files and Databases)

Use standard web server protection and configuration measures to protect the integrity of the server, databases, configuration files, and other operational components of the server.

5.3.3. Store Secrets in Secure Storage

There are different ways to store secrets of all kinds (tokens, client secrets) securely on a device or server.

Most multi-user operating systems segregate the personal storage of different system users. Moreover, most modern smartphone operating systems even support the storage of application-specific data in separate areas of file systems and protect the data from access by other applications. Additionally, applications can implement confidential data by using a user-supplied secret, such as a PIN or password.

Another option is to swap refresh token storage to a trusted backend server. This option in turn requires a resilient authentication mechanism between the client and backend server. Note: Applications should ensure that confidential data is kept confidential even after reading from secure storage, which typically means keeping this data in the local memory of the application.

5.3.4. Utilize Device Lock to Prevent Unauthorized Device Access

On a typical modern phone, there are many "device lock" options that can be utilized to provide additional protection when a device is stolen or misplaced. These include PINs, passwords, and other biometric features such as "face recognition". These are not equal in the level of security they provide.

5.3.5. Link the "state" Parameter to User Agent Session

The "state" parameter is used to link client requests and prevent CSRF attacks, for example, attacks against the redirect URI. An attacker could inject their own authorization "code" or access token, which can result in the client using an access token associated with the attacker's protected resources rather than the victim's (e.g., save the victim's bank account information to a protected resource controlled by the attacker).

The client should utilize the "state" request parameter to send the authorization server a value that binds the request to the user agent's authenticated state (e.g., a hash of the session cookie used to authenticate the user agent) when making an authorization request. Once authorization has been obtained from the end user, the authorization server redirects the end-user's user agent back to the client with the required binding value contained in the "state" parameter.

The binding value enables the client to verify the validity of the request by matching the binding value to the user agent's authenticated state.

5.4. Resource Servers

The following section details security considerations for resource servers.

5.4.1. Authorization Headers

Authorization headers are recognized and specially treated by HTTP proxies and servers. Thus, the usage of such headers for sending access tokens to resource servers reduces the likelihood of leakage or unintended storage of authenticated requests in general, and especially Authorization headers.

5.4.2. Authenticated Requests

An authorization server may bind tokens to a certain client identifier and enable resource servers to validate that association on resource access. This will require the resource server to authenticate the originator of a request as the legitimate owner of a particular token. There are several options to implement this countermeasure:

- The authorization server may associate the client identifier with the token (either internally or in the payload of a self-contained token). The client then uses client certificate-based HTTP authentication on the resource server's endpoint to authenticate its identity, and the resource server validates the name with the name referenced by the token.

- Same as the option above, but the client uses his private key to sign the request to the resource server (the public key is either contained in the token or sent along with the request).

- Alternatively, the authorization server may issue a token-bound key, which the client uses in a Holder-of-Key proof to authenticate the client's use of the token. The resource server obtains the secret directly from the authorization server, or the secret is contained in an encrypted section of the token. In that way, the resource server does not "know" the client but is able to validate whether the authorization server issued the token to that client.

Authenticated requests are a countermeasure against abuse of tokens by counterfeit resource servers.

5.4.3. Signed Requests

A resource server may decide to accept signed requests only, either to replace transport-level security measures or to complement such measures. Every signed request should be uniquely identifiable and should not be processed twice by the resource server. This countermeasure helps to mitigate:

- modifications of the message and

- replay attempts

5.5. A Word on User Interaction and User-Installed Apps

OAuth, as a security protocol, is distinctive in that its flow usually involves significant user interaction, making the end user a part of the security model. This creates some important difficulties in defending against some of the threats discussed above. Some of these points have already been made, but it's worth repeating and highlighting them here.

- End users must understand what they are being asked to approve (see Section 5.2.4.2). Users often do not have the expertise to understand the ramifications of saying "yes" to an authorization request and are likely not to be able to see subtle differences in the wording of requests. Malicious software can confuse the user, tricking the user into approving almost anything.

- End-user devices are prone to software compromise. This has been a long-standing problem, with frequent attacks on web browsers and other parts of the user's system. But with the increasing popularity of user-installed "apps", the threat posed by compromised or malicious end-user software is very strong and is one that is very difficult to mitigate.

- Be aware that users will demand to install and run such apps, and that compromised or malicious ones can steal credentials at many points in the data flow. They can intercept the very user login credentials that OAuth is designed to protect. They can request authorization far beyond what they have led the user to understand and approve. They can automate a response on behalf of the user, hiding the whole process. No solution is offered here, because none is known; this remains in the space between better security and better usability.

- Addressing these issues by restricting the use of user-installed software may be practical in some limited environments and can be used as a countermeasure in those cases. Such restrictions are not practical in the general case, and mechanisms for after-the-fact recovery should be in place.

- While end users are mostly incapable of properly vetting applications they load onto their devices, those who deploy authorization servers might have tools at their disposal to mitigate malicious clients. For example, a well-run authorization server must only assert client properties to the end user it is effectively capable of validating, explicitly point out which properties it cannot validate, and indicate to the end user the risk associated with granting access to the particular client.

6. Acknowledgements

We would like to thank Stephen Farrell, Barry Leiba, Hui-Lan Lu, Francisco Corella, Peifung E. Lam, Shane B. Weeden, Skylar Woodward, Niv Steingarten, Tim Bray, and James H. Manger for their comments and contributions.

7. References

7.1. Normative References

[RFC6749] Hardt, D., "The OAuth 2.0 Authorization Framework", RFC 6749, October 2012.

[RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, October 2012.

7.2. Informative References

[Framebusting]
Rydstedt, G., Bursztein, Boneh, D., and C. Jackson, "Busting Frame Busting: a Study of Clickjacking Vulnerabilities on Popular Sites", IEEE 3rd Web 2.0 Security and Privacy Workshop, May 2010, .

[IMEI] 3GPP, "International Mobile station Equipment Identities (IMEI)", 3GPP TS 22.016 11.0.0, September 2012, .

[OASIS.saml-core-2.0-os]
Cantor, S., Ed., Kemp, J., Ed., Philpott, R., Ed., and E. Maler, Ed., "Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0", OASIS Standard saml-core-2.0-os, March 2005, .

[OASIS.sstc-saml-bindings-1.1]
Maler, E., Ed., Mishra, P., Ed., and R. Philpott, Ed., "Bindings and Profiles for the OASIS Security Assertion Markup Language (SAML) V1.1", September 2003, .

[OASIS.sstc-sec-analysis-response-01]
Linn, J., Ed., and P. Mishra, Ed., "SSTC Response to "Security Analysis of the SAML Single Sign-on Browser/ Artifact Profile"", January 2005, .

[OAuth-ASSERTIONS]
Campbell, B., Mortimore, C., Jones, M., and Y. Goland, "Assertion Framework for OAuth 2.0", Work in Progress, December 2012.

[OAuth-HTTP-MAC]
Richer, J., Ed., Mills, W., Ed., and H. Tschofenig, Ed., "OAuth 2.0 Message Authentication Code (MAC) Tokens", Work in Progress, November 2012.

[OAuth-JWT]
Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", Work in Progress, December 2012.

[OAuth-REVOCATION]
Lodderstedt, T., Ed., Dronia, S., and M. Scurtescu, "Token Revocation", Work in Progress, November 2012.

[OPENID] "OpenID Foundation Home Page", .

[OWASP] "Open Web Application Security Project Home Page", .

[Portable-Contacts]
Smarr, J., "Portable Contacts 1.0 Draft C", August 2008, .

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.

[RFC4086] Eastlake, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, June 2005.

[RFC4120] Neuman, C., Yu, T., Hartman, S., and K. Raeburn, "The Kerberos Network Authentication Service (V5)", RFC 4120, July 2005.

[RFC4301] Kent, S. and K. Seo, "Security Architecture for the Internet Protocol", RFC 4301, December 2005.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

[SSL-Latency]
Sissel, J., Ed., "SSL handshake latency and HTTPS optimizations", June 2010.

[Sec-Analysis]
Gross, T., "Security Analysis of the SAML Single Sign-on Browser/Artifact Profile", 19th Annual Computer Security Applications Conference, Las Vegas, December 2003.

[X-Frame-Options]
Ross, D. and T. Gondrom, "HTTP Header X-Frame-Options", Work in Progress, October 2012.

[iFrame] World Wide Web Consortium, "Frames in HTML documents", W3C HTML 4.01, December 1999,
.

Authors' Addresses

Torsten Lodderstedt (editor)
Deutsche Telekom AG

EMail: torsten@lodderstedt.net

Mark McGloin
IBM

EMail: mark.mcgloin@ie.ibm.com

Phil Hunt
Oracle Corporation

EMail: phil.hunt@yahoo.com

Chapter 5

RFC 8252: OAuth 2.0 for Native and Mobile Apps

The intended audience for this spec is implementers of mobile apps or apps running on desktop devices, where interactions between the app and the browser are not as straightforward as in a browser-only environment. RFC 8252 is the first "Best Current Practice" (BCP) published by the group.

This is more of a set of guidelines rather than changes to the underlying protocol. It describes things like not allowing the applications to open an embedded web view which would be more susceptible to phishing attacks, as well as platform-specific recommendations on how to do so. It also requires that these types of apps use the PKCE extension since they are unable to use a client secret.

This was published in 2017, a full 5 years after the original RFC, and the mobile app landscape had matured a lot in those five years. This spec intends to bring the original RFC up to date based on the developments in mobile platforms in that time. That said, a few years later and things have continued to change, so some of the platform-specific details mentioned in this spec may now be out of date.

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

W. Denniss
Google
J. Bradley
Ping Identity

October 2017

OAuth 2.0 for Native Apps

Abstract

OAuth 2.0 authorization requests from native apps should only be made through external user-agents, primarily the user's browser. This specification details the security and usability reasons why this is the case and how native apps and authorization servers can implement this best practice.

Status of This Memo

This memo documents an Internet Best Current Practice.

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

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

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

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

1. Introduction

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

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

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

2. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Terminology

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4. Overview

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

Previously, it was common for native apps to use embedded user-agents (commonly implemented with web-views) for OAuth authorization requests. That approach has many drawbacks, including the host app being able to copy user credentials and cookies as well as the user needing to authenticate from scratch in each app. See Section 8.12

for a deeper analysis of the drawbacks of using embedded user-agents for OAuth.

Native app authorization requests that use the browser are more secure and can take advantage of the user's authentication state. Being able to use the existing authentication session in the browser enables single sign-on, as users don't need to authenticate to the authorization server each time they use a new app (unless required by the authorization server policy).

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

4.1. Authorization Flow for Native Apps Using the Browser

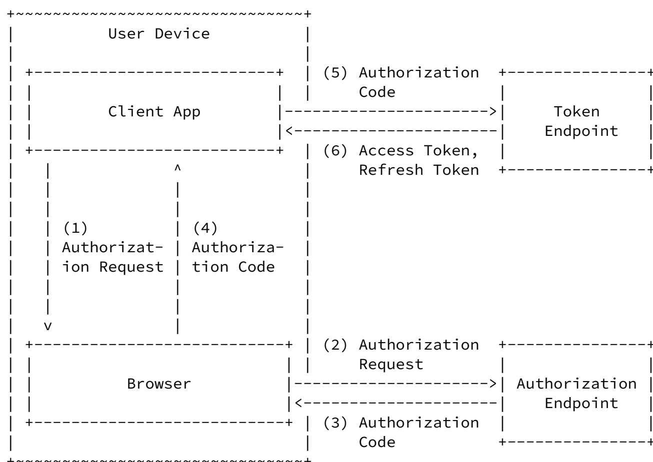


Figure 1: Native App Authorization via an External User-Agent

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

(1) Client app opens a browser tab with the authorization request.

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

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

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

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

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

5. Using Inter-App URI Communication for OAuth

Just as URIs are used for OAuth 2.0 [RFC6749] on the web to initiate the authorization request and return the authorization response to the requesting website, URIs can be used by native apps to initiate the authorization request in the device's browser and return the response to the requesting native app.

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

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

6. Initiating the Authorization Request from a Native App

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

The function of the redirect URI for a native app authorization request is similar to that of a web-based authorization request. Rather than returning the authorization response to the OAuth client's server, the redirect URI used by a native app returns the response to the app. Several options for a redirect URI that will return the authorization response to the native app in different platforms are documented in Section 7. Any redirect URI that allows the app to receive the URI and inspect its parameters is viable.

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

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

This best practice focuses on the browser as the RECOMMENDED external user-agent for native apps. An external user-agent designed specifically for user authorization and capable of processing authorization requests and responses like a browser MAY also be used. Other external user-agents, such as a native app provided by the authorization server may meet the criteria set out in this best practice, including using the same redirection URI properties, but their use is out of scope for this specification.

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

7. Receiving the Authorization Response in a Native App

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

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

7.1. Private-Use URI Scheme Redirection

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

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

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

For example, an app that controls the domain name "app.example.com" can use "com.example.app" as their scheme. Some authorization servers assign client identifiers based on domain names, for example, "client1234.usercontent.example.net", which can also be used as the domain name for the scheme when reversed in the same manner. A scheme such as "myapp", however, would not meet this requirement, as it is not based on a domain name.

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

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

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

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

7.2. Claimed "https" Scheme URI Redirection

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

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

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

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

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

7.3. Loopback Interface Redirection

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

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

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

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

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

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

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

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

8. Security Considerations

8.1. Protecting the Authorization Code

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

A limitation of using private-use URI schemes for redirect URIs is that multiple apps can typically register the same scheme, which makes it indeterminate as to which app will receive the authorization code. Section 1 of PKCE [RFC7636] details how this limitation can be used to execute a code interception attack.

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

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

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

Section 6 requires that both clients and servers use PKCE for public native app clients. Authorization servers SHOULD reject authorization requests from native apps that don't use PKCE by returning an error message, as defined in Section 4.4.1 of PKCE [RFC7636].

8.2. OAuth Implicit Grant Authorization Flow

The OAuth 2.0 implicit grant authorization flow (defined in Section 4.2 of OAuth 2.0 [RFC6749]) generally works with the practice of performing the authorization request in the browser and receiving the authorization response via URI-based inter-app communication. However, as the implicit flow cannot be protected by PKCE [RFC7636] (which is required in Section 8.1), the use of the Implicit Flow with native apps is NOT RECOMMENDED.

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

8.3. Loopback Redirect Considerations

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

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

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

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

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

8.4. Registration of Native App Clients

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

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

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

In addition to the collision-resistant properties, requiring a URI scheme based on a domain name that is under the control of the app can help to prove ownership in the event of a dispute where two apps claim the same private-use URI scheme (where one app is acting maliciously). For example, if two apps claimed "com.example.app", the owner of "example.com" could petition the app store operator to remove the counterfeit app. Such a petition is harder to prove if a generic URI scheme was used.

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

8.5. Client Authentication

Secrets that are statically included as part of an app distributed to multiple users should not be treated as confidential secrets, as one user may inspect their copy and learn the shared secret. For this reason, and those stated in Section 5.3.1 of [RFC6819], it is NOT RECOMMENDED for authorization servers to require client

authentication of public native apps clients using a shared secret, as this serves little value beyond client identification which is already provided by the "client_id" request parameter.

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

8.6. Client Impersonation

As stated in Section 10.2 of OAuth 2.0 [RFC6749], the authorization server SHOULD NOT process authorization requests automatically without user consent or interaction, except when the identity of the client can be assured. This includes the case where the user has previously approved an authorization request for a given client id -- unless the identity of the client can be proven, the request SHOULD be processed as if no previous request had been approved.

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

8.7. Fake External User-Agents

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

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

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

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

8.8. Malicious External User-Agents

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

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

8.9. Cross-App Request Forgery Protections

Section 5.3.5 of [RFC6819] recommends using the "state" parameter to link client requests and responses to prevent CSRF (Cross-Site Request Forgery) attacks.

To mitigate CSRF-style attacks over inter-app URI communication channels (so called "cross-app request forgery"), it is similarly RECOMMENDED that native apps include a high-entropy secure random number in the "state" parameter of the authorization request and reject any incoming authorization responses without a state value that matches a pending outgoing authorization request.

8.10. Authorization Server Mix-Up Mitigation

To protect against a compromised or malicious authorization server attacking another authorization server used by the same app, it is REQUIRED that a unique redirect URI is used for each authorization server used by the app (for example, by varying the path component), and that authorization responses are rejected if the redirect URI they were received on doesn't match the redirect URI in an outgoing authorization request.

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

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

8.11. Non-Browser External User-Agents

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

8.12. Embedded User-Agents

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

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

In typical web-view-based implementations of embedded user-agents, the host application can record every keystroke entered in the login form to capture usernames and passwords, automatically submit forms to bypass user consent, and copy session cookies and use them to perform authenticated actions as the user.

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

Encouraging users to enter credentials in an embedded user-agent without the usual address bar and visible certificate validation features that browsers have makes it impossible for the user to know if they are signing in to the legitimate site; even when they are, it trains them that it's OK to enter credentials without validating the site first.

Aside from the security concerns, embedded user-agents do not share the authentication state with other apps or the browser, requiring the user to log in for every authorization request, which is often considered an inferior user experience.

9. IANA Considerations

This document does not require any IANA actions.

Section 7.1 specifies how private-use URI schemes are used for inter-app communication in OAuth protocol flows. This document requires in Section 7.1 that such schemes are based on domain names owned or assigned to the app, as recommended in Section 3.8 of [RFC7595]. Per Section 6 of [RFC7595], registration of domain-based URI schemes with IANA is not required.

10. References

10.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, .

[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, .

[RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, .

[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, .

[RFC7595] Thaler, D., Ed., Hansen, T., and T. Hardie, "Guidelines and Registration Procedures for URI Schemes", BCP 35, RFC 7595, DOI 10.17487/RFC7595, June 2015, .

[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, .

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, .

10.2. Informative References

[RFC6819] Lodderstedt, T., Ed., McGloin, M., and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations", RFC 6819, DOI 10.17487/RFC6819, January 2013, .

[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, .

[AppAuth] OpenID Connect Working Group, "AppAuth", September 2017, .

[AppAuth.iOSmacOS] Wright, S., Denniss, W., et al., "AppAuth for iOS and macOS", February 2016, .

[AppAuth.Android] McGinniss, I., Denniss, W., et al., "AppAuth for Android", February 2016, .

[SamplesForWindows] Denniss, W., "OAuth for Apps: Samples for Windows", July 2016, .

Appendix A. Server Support Checklist

OAuth servers that support native apps must:

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

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

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

4. Not assume that native app clients can keep a secret. If secrets are distributed to multiple installs of the same native app, they should not be treated as confidential. See Section 8.5.

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

Appendix B. Platform-Specific Implementation Details

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

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

B.1. iOS Implementation Details

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

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

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

Claimed "https" scheme URIs are the preferred redirect choice on iOS 9 and above due to the ownership proof that is provided by the operating system.

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

B.2. Android Implementation Details

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

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

To receive the authorization response, private-use URI schemes are broadly supported through Android Implicit Intents. Claimed "https" scheme redirect URIs through Android App Links are available on Android 6.0 and above. Both types of redirect URIs are registered in the application's manifest.

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

B.3. Windows Implementation Details

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

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

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

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

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

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

B.4. macOS Implementation Details

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

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

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

B.5. Linux Implementation Details

Opening the authorization request in the user's default browser requires a distro-specific command: "xdg-open" is one such tool.

The loopback redirect is the recommended redirect choice for desktop apps on Linux to receive the authorization response. Apps SHOULD NOT set the "SO_REUSEPORT" or "SO_REUSEADDR" socket options in order to prevent other apps binding to the same socket.

Acknowledgements

The authors would like to acknowledge the work of Marius Scurtescu and Ben Wiley Sittler, whose design for using private-use URI schemes in native app OAuth 2.0 clients at Google formed the basis of Section 7.1.

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

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

Authors' Addresses

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

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

John Bradley
Ping Identity

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

Chapter 6

Draft: OAuth 2.0 for Browser-Based Apps

OAuth 2.0 for Browser-Based Apps describes security requirements and other recommendations for JavaScript apps (commonly known as Single-Page Apps) using OAuth.

As of this publication, this document is still in draft form and is not yet an RFC. It is likely to go through some more changes before it is finalized. That said, it has been adopted by the working group and gone through several rounds of changes, which means people broadly recognize the need for this kind of guidance.

This document is intended to be a complement to the Native App Best Current Practice, addressing the specifics of a browser-based environment instead.

Some of the concrete recommendations it provides are using the Authorization Code flow with PKCE instead of using the Implicit flow, and disallowing the Password grant by browser apps. It also provides a few different architectural patterns available to apps in this environment, and provides guidance on storing tokens in browsers.

Web Authorization Protocol
Internet-Draft
Intended status: Best Current Practice
Expires: 17 March 2023

A. Parecki
Okta
D. Waite
Ping Identity

13 September 2022

OAuth 2.0 for Browser-Based Apps
draft-ietf-oauth-browser-based-apps-11

Abstract

This specification details the security considerations and best practices that must be taken into account when developing browser-based applications that use OAuth 2.0.

Discussion Venues

This note is to be removed before publishing as an RFC.

Discussion of this document takes place on the Web Authorization Protocol Working Group mailing list (oauth@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/oauth/.

Source for this draft and an issue tracker can be found at https://github.com/oauth-wg/oauth-browser-based-apps.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 17 March 2023.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction 3
2. Notational Conventions 3
3. Terminology 3
4. Overview 4
5. First-Party Applications 5
6. Application Architecture Patterns 6
6.1. Single-Domain Browser-Based Apps (not using OAuth) . . . 6
6.2. Backend For Frontend (BFF) Proxy 7
6.2.1. Security considerations 9
6.3. Token Mediating Backend 9
6.3.1. Security Considerations 11
6.4. JavaScript Applications obtaining tokens directly 11
6.4.1. Storing Tokens in Local or Session Storage 13
6.4.2. Service Worker as the OAuth Client 13
7. Authorization Code Flow 15
7.1. Initiating the Authorization Request from a Browser-Based Application 15
7.2. Handling the Authorization Code Redirect 16
8. Refresh Tokens 16
9. Security Considerations 17
9.1. Cross-Site Scripting Attacks (XSS) 17
9.2. Reducing the Impact of Token Exfiltration 18
9.3. Registration of Browser-Based Apps 18
9.4. Client Authentication 18
9.5. Client Impersonation 19
9.6. Cross-Site Request Forgery Protections 19
9.7. Authorization Server Mix-Up Mitigation 19
9.8. Cross-Domain Requests 20
9.9. Content Security Policy 20
9.10. OAuth Implicit Flow 21
9.10.1. Attacks on the Implicit Flow 21
9.10.2. Countermeasures 22
9.10.3. Disadvantages of the Implicit Flow 22
9.10.4. Historic Note 23
9.11. Additional Security Considerations 24
10. IANA Considerations 24
11. References 24

11.1. Normative References 24
11.2. Informative References 25
Appendix A. Server Support Checklist 25
Appendix B. Document History 26
Appendix C. Acknowledgements 29
Authors' Addresses 29

1. Introduction

This specification describes the current best practices for implementing OAuth 2.0 authorization flows in applications executing in a browser.

For native application developers using OAuth 2.0 and OpenID Connect, an IETF BCP (best current practice) was published that guides integration of these technologies. This document is formally known as [RFC8252] or BCP 212, but nicknamed "AppAuth" after the OpenID Foundation-sponsored set of libraries that assist developers in adopting these practices. [RFC8252] makes specific recommendations for how to securely implement OAuth in native applications, including incorporating additional OAuth extensions where needed.

OAuth 2.0 for Browser-Based Apps addresses the similarities between implementing OAuth for native apps and browser-based apps, and includes additional considerations when running in a browser. This is primarily focused on OAuth, except where OpenID Connect provides additional considerations.

Many of these recommendations are derived from the OAuth 2.0 Security Best Current Practice [oauth-security-topics] and browser-based apps are expected to follow those recommendations as well. This draft expands on and further restricts various recommendations in [oauth-security-topics].

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 [RFC2119].

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] and [RFC6750].

"Browser-based application": An application that is dynamically downloaded and executed in a web browser, usually written in JavaScript. Also sometimes referred to as a "single-page application", or "SPA".

4. Overview

At the time that OAuth 2.0 [RFC6749] and [RFC6750] were created, browser-based JavaScript applications needed a solution that strictly complied with the same-origin policy. Common deployments of OAuth 2.0 involved an application running on a different domain than the authorization server, so it was historically not possible to use the Authorization Code flow which would require a cross-origin POST request. This was one of the motivations for the definition of the Implicit flow, which returns the access token in the front channel via the fragment part of the URL, bypassing the need for a cross-origin POST request.

However, there are several drawbacks to the Implicit flow, generally involving vulnerabilities associated with the exposure of the access token in the URL. See Section 9.10 for an analysis of these attacks and the drawbacks of using the Implicit flow in browsers. Additional attacks and security considerations can be found in [oauth-security-topics].

In recent years, widespread adoption of Cross-Origin Resource Sharing (CORS), which enables exceptions to the same-origin policy, allows browser-based apps to use the OAuth 2.0 Authorization Code flow and make a POST request to exchange the authorization code for an access token at the token endpoint. In this flow, the access token is never exposed in the less-secure front channel. Furthermore, adding PKCE to the flow ensures that even if an authorization code is intercepted, it is unusable by an attacker.

For this reason, and from other lessons learned, the current best practice for browser-based applications is to use the OAuth 2.0 Authorization Code flow with PKCE.

Browser-based applications:

* MUST use the OAuth 2.0 Authorization Code flow with the PKCE extension when obtaining an access token

* MUST Protect themselves against CSRF attacks by either:

  - ensuring the authorization server supports PKCE, or

- by using the OAuth 2.0 "state" parameter or the OpenID Connect "nonce" parameter to carry one-time use CSRF tokens

* MUST Register one or more redirect URIs, and use only exact registered redirect URIs in authorization requests

OAuth 2.0 authorization servers supporting browser-based applications:

* MUST Require exact matching of registered redirect URIs

* MUST Support the PKCE extension

* MUST NOT issue access tokens in the authorization response

* If issuing refresh tokens to browser-based applications, then:

  - MUST rotate refresh tokens on each use or use sender-constrained refresh tokens, and

  - MUST set a maximum lifetime on refresh tokens or expire if they are not used in some amount of time

5. First-Party Applications

While OAuth was initially created to allow third-party applications to access an API on behalf of a user, it has proven to be useful in a first-party scenario as well. First-party apps are applications where the same organization provides both the API and the application.

Examples of first-party applications are a web email client provided by the operator of the email account, or a mobile banking application created by bank itself. (Note that there is no requirement that the application actually be developed by the same company; a mobile banking application developed by a contractor that is branded as the bank's application is still considered a first-party application.) The first-party app consideration is about the user's relationship to the application and the service.

To conform to this best practice, first-party applications using OAuth or OpenID Connect MUST use a redirect-based flow (such as the OAuth Authorization Code flow) as described later in this document.

The resource owner password credentials grant MUST NOT be used, as described in [oauth-security-topics] Section 2.4. Instead, by using the Authorization Code flow and redirecting the user to the authorization server, this provides the authorization server the

opportunity to prompt the user for multi-factor authentication options, take advantage of single sign-on sessions, or use third-party identity providers. In contrast, the resource owner password credentials grant does not provide any built-in mechanism for these, and would instead be extended with custom code.

6. Application Architecture Patterns

Here are the main architectural patterns available when building browser-based applications.

* single-domain, not using OAuth
* a JavaScript application with a stateful backend component
  - storing tokens and proxying all requests (BFF Proxy)
  - obtaining tokens and passing them to the frontend (Token Mediating Backend)
* a JavaScript application obtaining access tokens
  - via JavaScript code executed in the DOM
  - through a service worker

These architectures have different use cases and considerations.

6.1. Single-Domain Browser-Based Apps (not using OAuth)

For simple system architectures, such as when the JavaScript application is served from a domain that can share cookies with the domain of the API (resource server) and the authorization server, OAuth adds additional attack vectors that could be avoided with a different solution.

In particular, using any redirect-based mechanism of obtaining an access token enables the redirect-based attacks described in [oauth-security-topics] Section 4, but if the application, authorization server and resource server share a domain, then it is unnecessary to use a redirect mechanism to communicate between them.

An additional concern with handling access tokens in a browser is that in case of successful cross-site scripting (XSS) attack, tokens could be read and further used or transmitted by the injected code if no secure storage mechanism is in place.

As such, it could be considered to use an HTTP-only cookie between the JavaScript application and API so that the JavaScript code can't access the cookie value itself. The Secure cookie attribute should be used to ensure the cookie is not included in unencrypted HTTP requests. Additionally, the SameSite cookie attribute can be used to counter some CSRF attacks, but should not be considered the extent of the CSRF protection, as described in [draft-ietf-httpbis-rfc6265bis]

OAuth was originally created for third-party or federated access to APIs, so it may not be the best solution in a common-domain deployment. That said, there are still some advantages in using OAuth even in a common-domain architecture:

* Allows more flexibility in the future, such as if you were to later add a new domain to the system. With OAuth already in place, adding a new domain wouldn't require any additional rearchitecting.

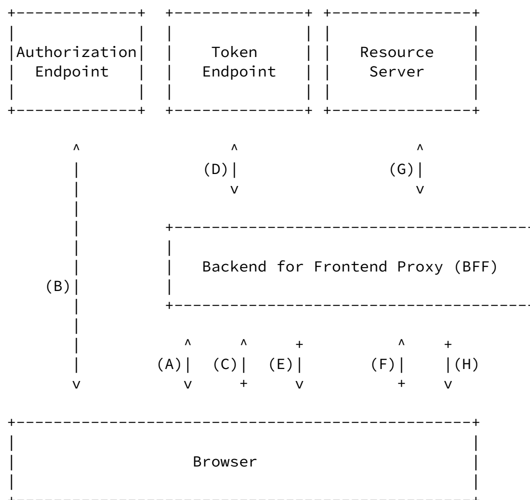
* Being able to take advantage of existing library support rather than writing bespoke code for the integration.

* Centralizing login and multifactor support, account management, and recovery at the OAuth server, rather than making it part of the application logic.

* Splitting of responsibilities between authenticating a user and serving resources

Using OAuth for browser-based apps in a first-party same-domain scenario provides these advantages, and can be accomplished by any of the architectural patterns described below.

6.2. Backend For Frontend (BFF) Proxy



In this architecture, commonly referred to as "backend for frontend" or "BFF", the JavaScript code is loaded from a dynamic BFF Proxy (A) that has the ability to execute code and handle the full authentication flow itself. This enables the ability to keep the call to actually get an access token outside the JavaScript application.

Note that this BFF Proxy is not the Resource Server, it is the OAuth client and would be accessing data at a separate resource server.

In this case, the BFF Proxy initiates the OAuth flow itself, by redirecting the browser to the authorization endpoint (B). When the user is redirected back, the browser delivers the authorization code to the BFF Proxy (C), where it can then exchange it for an access token at the token endpoint (D) using its client secret and PKCE code verifier. The BFF Proxy then keeps the access token and refresh token stored internally, and creates a separate session with the browser-based app via a traditional browser cookie (E).

When the JavaScript application in the browser wants to make a request to the Resource Server, it instead makes the request to the BFF Proxy (F), and the BFF Proxy will make the request with the access token to the Resource Server (G), and forward the response (H) back to the browser.

(Common examples of this architecture are an Angular front-end with a .NET backend, or a React front-end with a Spring Boot backend.)

The BFF Proxy SHOULD be considered a confidential client, and issued its own client secret. The BFF Proxy SHOULD use the OAuth 2.0 Authorization Code grant with PKCE to initiate a request for an access token. Detailed recommendations for confidential clients can be found in [oauth-security-topics] Section 2.1.1.

In this scenario, the connection between the browser and BFF Proxy SHOULD be a session cookie provided by the BFF Proxy.

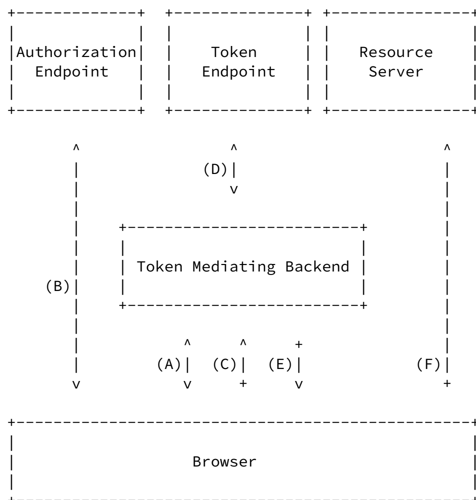
6.2.1. Security considerations

Security of the connection between code running in the browser and this BFF Proxy is assumed to utilize browser-level protection mechanisms. Details are out of scope of this document, but many recommendations can be found in the OWASP Cheat Sheet series (https://cheatsheetseries.owasp.org/), such as setting an HTTP-only and Secure cookie to authenticate the session between the browser and BFF Proxy. Additionally, cookies MUST be protected from leakage by other means, such as logs.

In this architecture, tokens are never sent to the front-end and are never accessible by any JavaScript code, so it fully protects against XSS attackers stealing tokens. However, an XSS attacker may still be able to make authenticated requests to the BFF Proxy which will in turn make requests to the resource server including the user's legitimate token. While the attacker is unable to extract and use the access token elsewhere, they can still effectively make authenticated requests to the resource server.

6.3. Token Mediating Backend

An alternative to a full BFF where all resource requests go through the backend is to use a token mediating backend which obtains the tokens and then forwards the tokens to the browser.



The frontend code makes a request to the Token Mediating Backend (A), and the backend initiates the OAuth flow itself, by redirecting the browser to the authorization endpoint (B). When the user is redirected back, the browser delivers the authorization code to the application server (C), where it can then exchange it for an access token at the token endpoint (D) using its client secret and PKCE code verifier. The backend delivers the tokens to the browser (E), which stores them for later use. The browser makes requests to the resource server directly (F) including the token it has stored.

The main advantage this architecture provides over the full BFF architecture previously described is that the backend service is only involved in the acquisition of tokens, and doesn't have to proxy every request in the future. Routing every API call through a backend can be expensive in terms of performance and latency, and can create challenges in deploying the application across many regions. Instead, routing only the token acquisition through a backend means fewer requests are made to the backend. This improves the performance and reduces the latency of requests from the frontend, and reduces the amount of infrastructure needed in the backend.

Similar to the previously described BFF Proxy pattern, The Token Mediating Backend SHOULD be considered a confidential client, and issued its own client secret. The Token Mediating Backend SHOULD use the OAuth 2.0 Authorization Code grant with PKCE to initiate a request for an access token. Detailed recommendations for confidential clients can be found in [oauth-security-topics] Section 2.1.1.

In this scenario, the connection between the browser and Token Mediating Backend SHOULD be a session cookie provided by the backend.

The Token Mediating Backend SHOULD cache tokens it obtains from the authorization server such that when the frontend needs to obtain new tokens, it can do so without the additional round trip to the authorization server if the tokens are still valid.

The frontend SHOULD NOT persist tokens in local storage or similar mechanisms; instead, the frontend SHOULD store tokens only in memory, and make a new request to the backend if no tokens exist. This provides fewer attack vectors for token exfiltration should an XSS attack be successful.

Editor's Note: A method of implementing this architecture is described by the [tmi-bff] draft, although it is currently an expired individual draft and has not been proposed for adoption to the OAuth Working Group.

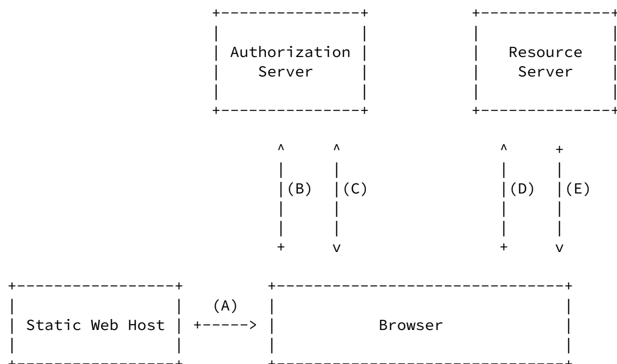
6.3.1. Security Considerations

If the backend caches tokens from the authorization server, it presents scopes elevation risks if applied indiscriminately. If the token cached by the authorization server features a superset of the scopes requested by the frontend, the backend SHOULD NOT return it to the frontend; instead it SHOULD perform a new request with the smaller set of scopes to the authorization server.

In the case of a successful XSS attack, the attacker may be able to access the tokens if the tokens are persisted in the frontend, but is less likely to be able to access the tokens if they are stored only in memory. However, a successful XSS attack will also allow the attacker to call the Token Mediating Backend itself to retrieve the cached token or start a new OAuth flow.

6.4. JavaScript Applications obtaining tokens directly

This section describes the architecture of a JavaScript application obtaining tokens from the authorization itself, with no intermediate proxy server.



In this architecture, the JavaScript code is first loaded from a static web host into the browser (A), and the application then runs in the browser. This application is considered a public client, since there is no way to issue it a client secret in this model.

The code in the browser initiates the Authorization Code flow with the PKCE extension (described in Section 7) (B) above, and obtains an access token via a POST request (C).

The application is then responsible for storing the access token (and optional refresh token) as securely as possible using appropriate browser APIs.

When the JavaScript application in the browser wants to make a request to the Resource Server, it can interact with the Resource Server directly. It includes the access token in the request (D) and receives the Resource Server's response (E).

In this scenario, the Authorization Server and Resource Server MUST support the necessary CORS headers to enable the JavaScript code to make these POST requests from the domain on which the script is executing. (See Section 9.8 for additional details.)

Besides the general risks of XSS, if tokens are stored or handled by the browser, XSS poses an additional risk of token exfiltration. In this architecture, the JavaScript application is storing the access token so that it can make requests directly to the resource server. There are two primary methods by which the application can store the token, with different security considerations of each.

6.4.1. Storing Tokens in Local or Session Storage

If the JavaScript in the DOM will be making requests directly to the resource server, the simplest mechanism is to store the tokens somewhere accessible to the DOM.

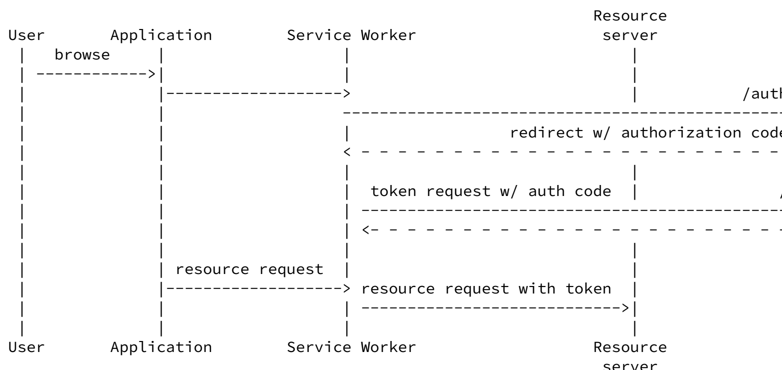
In case of a successful XSS attack, the injected code will have full access to the stored tokens and can exfiltrate them to the attacker.

6.4.2. Service Worker as the OAuth Client

In this scenario, a Service Worker (https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API) is responsible for obtaining tokens from the authorization server and making requests to the resource server.

Service workers are run in a separate context from the DOM, have no access to the DOM, and the DOM has no access to the service worker. This makes service workers the most secure place to store tokens, as an XSS attack is unable to exfiltrate the tokens.

In this architecture, a service worker intercepts calls from the frontend to the resource server. As such, it completely isolates calls to the authorization server from XSS attack surface, as all tokens are safely kept in the service worker context without any access from other JavaScript contexts. The service worker is then solely responsible for adding the token in the authorization header to calls to the resource server.



6.4.2.1. Implementation Guidelines

* The service worker MUST initiate the OAuth 2.0 Authorization Code grant with PKCE itself.

* The service worker MUST intercept the authorization code when the _authorization server_ redirects to the application.

* The service worker implementation MUST then initiate the token request itself.

* The service worker MUST not transmit tokens, authorization codes or PKCE secrets (e.g. code verifier) to the frontend application.

* The service worker MUST block authorization requests and token requests initiating from the frontend application in order to avoid any front-end side-channel for getting credentials: the only way of starting the authorization flow is through the service worker. This protects against re-authorization from XSS-injected code.

* The application MUST register the Service Worker before running any code interacting with the user.

6.4.2.2. Security Considerations

A successful XSS attack on an application using this Service Worker pattern would be unable to exfiltrate existing tokens stored by the application. However, an XSS attacker may still be able to cause the Service Worker to make authenticated requests to the resource server including the user's legitimate token.

In case of a vulnerability leading to the Service Worker not being registered, an XSS attack would result in the attacker being able to initiate a new OAuth flow to obtain new tokens itself.

To prevent the Service Worker from being unregistered, the Service Worker registration must happen as first step of the application start, and before any user interaction. Starting the Service worker before the rest of the application, and the fact that there is no way to remove a Service Worker from an active application (https://www.w3.org/TR/service-workers/#navigator-service-worker-unregister), reduces the risk of an XSS attack being able to prevent the Service Worker from being registered.

7. Authorization Code Flow

Browser-based applications that are public clients and use the Authorization Code grant type described in Section 4.1 of OAuth 2.0 [RFC6749] MUST also follow these additional requirements described in this section.

7.1. Initiating the Authorization Request from a Browser-Based Application

Browser-based applications that are public clients MUST implement the Proof Key for Code Exchange (PKCE [RFC7636]) extension when obtaining an access token, and authorization servers MUST support and enforce PKCE for such clients.

The PKCE extension prevents an attack where the authorization code is intercepted and exchanged for an access token by a malicious client, by providing the authorization server with a way to verify the client instance that exchanges the authorization code is the same one that initiated the flow.

Browser-based applications MUST prevent CSRF attacks against their redirect URI. This can be accomplished by any of the below:

* using PKCE, and confirming that the authorization server supports PKCE

* using a unique value for the OAuth 2.0 "state" parameter

* if the application is using OpenID Connect, by using the OpenID Connect "nonce" parameter

7.2. Handling the Authorization Code Redirect

Authorization servers MUST require an exact match of a registered redirect URI. As described in [oauth-security-topics] Section 4.1.1. this helps to prevent attacks targeting the authorization code.

8. Refresh Tokens

Refresh tokens provide a way for applications to obtain a new access token when the initial access token expires. With public clients, the risk of a leaked refresh token is greater than leaked access tokens, since an attacker may be able to continue using the stolen refresh token to obtain new access tokens potentially without being detectable by the authorization server.

Javascript-accessible storage mechanisms like _Local Storage_ provide an attacker with several opportunities by which a refresh token can be leaked, just as with access tokens. As such, these mechanisms are considered a higher risk for handling refresh tokens.

Authorization servers may choose whether or not to issue refresh tokens to browser-based applications. [oauth-security-topics] describes some additional requirements around refresh tokens on top of the recommendations of [RFC6749]. Applications and authorization servers conforming to this BCP MUST also follow the recommendations in [oauth-security-topics] around refresh tokens if refresh tokens are issued to browser-based applications.

In particular, authorization servers:

* MUST either rotate refresh tokens on each use OR use sender-constrained refresh tokens as described in [oauth-security-topics] Section 4.13.2

* MUST either set a maximum lifetime on refresh tokens OR expire if the refresh token has not been used within some amount of time

* MUST NOT extend the lifetime of the new refresh token beyond the lifetime of the initial refresh token

* upon issuing a rotated refresh token, MUST NOT extend the lifetime of the new refresh token beyond the lifetime of the initial refresh token if the refresh token has a preestablished expiration time

For example:

* A user authorizes an application, issuing an access token that lasts 1 hour, and a refresh token that lasts 24 hours

* After 1 hour, the initial access token expires, so the application uses the refresh token to get a new access token

* The authorization server returns a new access token that lasts 1 hour, and a new refresh token that lasts 23 hours

* This continues until 24 hours pass from the initial authorization

* At this point, when the application attempts to use the refresh token after 24 hours, the request will fail and the application will have to involve the user in a new authorization request

By limiting the overall refresh token lifetime to the lifetime of the initial refresh token, this ensures a stolen refresh token cannot be used indefinitely.

Authorization servers MAY set different policies around refresh token issuance, lifetime and expiration for browser-based applications compared to other public clients.

9. Security Considerations

9.1. Cross-Site Scripting Attacks (XSS)

For all known architectures, all precautions MUST be taken to prevent cross-site scripting (XSS) attacks. In general, XSS attacks are a huge risk, and can lead to full compromise of the application.

If tokens are handled or accessible by the browser, there is a risk that a XSS attack can lead to token exfiltration.

Even if tokens are never sent to the frontend and are never accessible by any JavaScript code, an XSS attacker may still be able to make authenticated requests to the resource server by mimicking legitimate code in the DOM. For example, the attacker may make a request to the BFF Proxy which will in turn make requests to the resource server including the user's legitimate token. In the Service Worker example, the attacker may make an API call to the

Service Worker which will then turn around and make a request to the resource server with the legitimate token. While the attacker is unable to extract and use the access token elsewhere, they can still effectively make authenticated requests to the resource server to steal or modify data.

9.2. Reducing the Impact of Token Exfiltration

If tokens are ever accessible to the browser or to any JavaScript code, there is always a risk of token exfiltration. The particular risk may change depending on the architecture chosen. Regardless of the particular architecture chosen, these additional security considerations limit the impact of token exfiltration:

* The authorization server SHOULD restrict access tokens to strictly needed resources, to avoid escalating the scope of the attack.

* To avoid information disclosure from ID Tokens, the authorization server SHOULD NOT include any ID token claims that aren't used by the frontend.

* Refresh tokens should be used in accordance with the guidance in Section 8.

9.3. Registration of Browser-Based Apps

Browser-based applications (with no backend) are considered public clients as defined by Section 2.1 of OAuth 2.0 [RFC6749], and MUST be registered with the authorization server as such. Authorization servers MUST record the client type in the client registration details in order to identify and process requests accordingly.

Authorization servers MUST require that browser-based applications register one or more redirect URIs.

9.4. Client Authentication

Since a browser-based application's source code is delivered to the end-user's browser, it cannot contain provisioned secrets. As such, a browser-based app with native OAuth support is considered a public client as defined by Section 2.1 of OAuth 2.0 [RFC6749].

Secrets that are statically included as part of an app distributed to multiple users should not be treated as confidential secrets, as one user may inspect their copy and learn the shared secret. For this reason, and those stated in Section 5.3.1 of [RFC6819], it is NOT RECOMMENDED for authorization servers to require client authentication of browser-based applications using a shared secret, as this serves little value beyond client identification which is already provided by the client_id request parameter.

Authorization servers that still require a statically included shared secret for SPA clients MUST treat the client as a public client, and not accept the secret as proof of the client's identity. Without additional measures, such clients are subject to client impersonation (see Section 9.5 below).

9.5. Client Impersonation

As stated in Section 10.2 of OAuth 2.0 [RFC6749], the authorization server SHOULD NOT process authorization requests automatically without user consent or interaction, except when the identity of the client can be assured.

If authorization servers restrict redirect URIs to a fixed set of absolute HTTPS URIs, preventing the use of wildcard domains, wildcard paths, or wildcard query string components, this exact match of registered absolute HTTPS URIs MAY be accepted by authorization servers as proof of identity of the client for the purpose of deciding whether to automatically process an authorization request when a previous request for the client_id has already been approved.

9.6. Cross-Site Request Forgery Protections

Clients MUST prevent Cross-Site Request Forgery (CSRF) attacks against their redirect URI. Clients can accomplish this by either ensuring the authorization server supports PKCE and relying on the CSRF protection that PKCE provides, or if the client is also an OpenID Connect client, using the OpenID Connect "nonce" parameter, or by using the "state" parameter to carry one-time-use CSRF tokens as described in Section 7.1.

See Section 2.1 of [oauth-security-topics] for additional details.

9.7. Authorization Server Mix-Up Mitigation

Authorization server mix-up attacks mark a severe threat to every client that supports at least two authorization servers. To conform to this BCP such clients MUST apply countermeasures to defend against mix-up attacks.

It is RECOMMENDED to defend against mix-up attacks by identifying and validating the issuer of the authorization response. This can be achieved either by using the "iss" response parameter, as defined in [oauth-iss-auth-resp], or by using the "iss" Claim of the ID token when OpenID Connect is used.

Alternative countermeasures, such as using distinct redirect URIs for each issuer, SHOULD only be used if identifying the issuer as described is not possible.

Section 4.4 of [oauth-security-topics] provides additional details about mix-up attacks and the countermeasures mentioned above.

9.8. Cross-Domain Requests

To complete the Authorization Code flow, the browser-based application will need to exchange the authorization code for an access token at the token endpoint. If the authorization server provides additional endpoints to the application, such as metadata URLs, dynamic client registration, revocation, introspection, discovery or user info endpoints, these endpoints may also be accessed by the browser-based app. Since these requests will be made from a browser, authorization servers MUST support the necessary CORS headers (defined in [Fetch]) to allow the browser to make the request.

This specification does not include guidelines for deciding whether a CORS policy for the token endpoint should be a wildcard origin or more restrictive. Note, however, that the browser will attempt to GET or POST to the API endpoint before knowing any CORS policy; it simply hides the succeeding or failing result from JavaScript if the policy does not allow sharing.

9.9. Content Security Policy

A browser-based application that wishes to use either long-lived refresh tokens or privileged scopes SHOULD restrict its JavaScript execution to a set of statically hosted scripts via a Content Security Policy ([CSP2]) or similar mechanism. A strong Content Security Policy can limit the potential attack vectors for malicious JavaScript to be executed on the page.

9.10. OAuth Implicit Flow

The OAuth 2.0 Implicit flow (defined in Section 4.2 of OAuth 2.0 [RFC6749]) works by the authorization server issuing an access token in the authorization response (front channel) without the code exchange step. In this case, the access token is returned in the fragment part of the redirect URI, providing an attacker with several opportunities to intercept and steal the access token.

Authorization servers MUST NOT issue access tokens in the authorization response, and MUST issue access tokens only from the token endpoint.

9.10.1. Attacks on the Implicit Flow

Many attacks on the Implicit flow described by [RFC6819] and Section 4.1.2 of [oauth-security-topics] do not have sufficient mitigation strategies. The following sections describe the specific attacks that cannot be mitigated while continuing to use the Implicit flow.

9.10.1.1. Threat: Manipulation of the Redirect URI

If an attacker is able to cause the authorization response to be sent to a URI under their control, they will directly get access to the authorization response including the access token. Several methods of performing this attack are described in detail in [oauth-security-topics].

9.10.1.2. Threat: Access Token Leak in Browser History

An attacker could obtain the access token from the browser's history. The countermeasures recommended by [RFC6819] are limited to using short expiration times for tokens, and indicating that browsers should not cache the response. Neither of these fully prevent this attack, they only reduce the potential damage.

Additionally, many browsers now also sync browser history to cloud services and to multiple devices, providing an even wider attack surface to extract access tokens out of the URL.

This is discussed in more detail in Section 4.3.2 of [oauth-security-topics].

9.10.1.3. Threat: Manipulation of Scripts

An attacker could modify the page or inject scripts into the browser through various means, including when the browser's HTTPS connection is being intercepted by, for example, a corporate network. While man-in-the-middle attacks are typically out of scope of basic security recommendations to prevent, in the case of browser-based apps they are much easier to perform. An injected script can enable an attacker to have access to everything on the page.

The risk of a malicious script running on the page may be amplified when the application uses a known standard way of obtaining access tokens, namely that the attacker can always look at the window.location variable to find an access token. This threat profile is different from an attacker specifically targeting an individual application by knowing where or how an access token obtained via the Authorization Code flow may end up being stored.

9.10.1.4. Threat: Access Token Leak to Third-Party Scripts

It is relatively common to use third-party scripts in browser-based apps, such as analytics tools, crash reporting, and even things like a Facebook or Twitter "like" button. In these situations, the author of the application may not be able to be fully aware of the entirety of the code running in the application. When an access token is returned in the fragment, it is visible to any third-party scripts on the page.

9.10.2. Countermeasures

In addition to the countermeasures described by [RFC6819] and [oauth-security-topics], using the Authorization Code flow with PKCE extension prevents the attacks described above by avoiding returning the access token in the redirect response at all.

When PKCE is used, if an authorization code is stolen in transport, the attacker is unable to do anything with the authorization code.

9.10.3. Disadvantages of the Implicit Flow

There are several additional reasons the Implicit flow is disadvantageous compared to using the standard Authorization Code flow.

* OAuth 2.0 provides no mechanism for a client to verify that a particular access token was intended for that client, which could lead to misuse and possible impersonation attacks if a malicious party hands off an access token it retrieved through some other means to the client.

* Returning an access token in the front-channel redirect gives the authorization server no assurance that the access token will actually end up at the application, since there are many ways this redirect may fail or be intercepted.

* Supporting the Implicit flow requires additional code, more upkeep and understanding of the related security considerations, while limiting the authorization server to just the Authorization Code flow reduces the attack surface of the implementation.

* If the JavaScript application gets wrapped into a native app, then [RFC8252] also requires the use of the Authorization Code flow with PKCE anyway.

In OpenID Connect, the ID Token is sent in a known format (as a JWT), and digitally signed. Returning an ID token using the Implicit flow (response_type=id_token) requires the client validate the JWT signature, as malicious parties could otherwise craft and supply fraudulent ID tokens. Performing OpenID Connect using the Authorization Code flow provides the benefit of the client not needing to verify the JWT signature, as the ID token will have been fetched over an HTTPS connection directly from the authorization server. Additionally, in many cases an application will request both an ID token and an access token, so it is simplier and provides fewer attack vectors to obtain both via the Authorization Code flow.

9.10.4. Historic Note

Historically, the Implicit flow provided an advantage to browser-based apps since JavaScript could always arbitrarily read and manipulate the fragment portion of the URL without triggering a page reload. This was necessary in order to remove the access token from the URL after it was obtained by the app.

Modern browsers now have the Session History API (described in "Session history and navigation" of [HTML]), which provides a mechanism to modify the path and query string component of the URL without triggering a page reload. This means modern browser-based apps can use the unmodified OAuth 2.0 Authorization Code flow, since they have the ability to remove the authorization code from the query string without triggering a page reload thanks to the Session History API.

9.11. Additional Security Considerations

The OWASP Foundation (https://www.owasp.org/) maintains a set of security recommendations and best practices for web applications, and it is RECOMMENDED to follow these best practices when creating an OAuth 2.0 Browser-Based application.

10. IANA Considerations

This document does not require any IANA actions.

11. References

11.1. Normative References

[CSP2] West, M., "Content Security Policy", October 2018.

[draft-ietf-httpbis-rfc6265bis]
Chen, L., Englehardt, S., West, M., and J. Wilander, "Cookies: HTTP State Management Mechanism", October 2021.

[Fetch] whatwg, "Fetch", 2018.

[oauth-iss-auth-resp]
Meyer zu Selhausen, K. and D. Fett, "OAuth 2.0 Authorization Server Issuer Identifier in Authorization Response", January 2021.

[oauth-security-topics]
Lodderstedt, T., Bradley, J., Labunets, A., and D. Fett, "OAuth 2.0 Security Best Current Practice", April 2021.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, .

[RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, .

[RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, .

[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, .

[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, .

[RFC8252] Denniss, W. and J. Bradley, "OAuth 2.0 for Native Apps", BCP 212, RFC 8252, DOI 10.17487/RFC8252, October 2017, .

11.2. Informative References

[HTML] whatwg, "HTML", 2020.

[tmi-bff] Bertocci, V. and B. Cambpell, "Token Mediating and session Information Backend For Frontend", April 2021.

Appendix A. Server Support Checklist

OAuth authorization servers that support browser-based apps MUST:

1. Require "https" scheme redirect URIs.

2. Require exact matching of registered redirect URIs.

3. Support PKCE [RFC7636]. Required to protect authorization code grants sent to public clients. See Section 7.1

4. Support cross-domain requests at the token endpoint in order to allow browsers to make the authorization code exchange request. See Section 9.8

5. Not assume that browser-based clients can keep a secret, and SHOULD NOT issue secrets to applications of this type.

6. Not support the Resource Owner Password grant for browser-based clients.

7. Follow the [oauth-security-topics] recommendations on refresh tokens, as well as the additional requirements described in Section 8.

Appendix B. Document History

[[To be removed from the final specification]]

-11

* Added a new architecture pattern: Token Mediating Backend

* Revised and added clarifications for the Service Worker pattern

* Editorial improvements in descriptions of the different architectures

* Rephrased headers

-10

* Revised the names of the architectural patterns

* Added a new pattern using a service worker as the OAuth client to manage tokens

* Added some considerations when storing tokens in Local or Session Storage

-09

* Provide additional context for the same-domain architecture pattern

* Added reference to draft-ietf-httpbis-rfc6265bis to clarify that SameSite is not the only CSRF protection measure needed

* Editorial improvements

-08

* Added a note to use the "Secure" cookie attribute in addition to SameSite etc

* Updates to bring this draft in sync with the latest Security BCP

* Updated text for mix-up countermeasures to reference the new "iss" extension

* Changed "SHOULD" for refresh token rotation to MUST either use rotation or sender-constraining to match the Security BCP

* Fixed references to other specs and extensions

* Editorial improvements in descriptions of the different architectures

-07

* Clarify PKCE requirements apply only to issuing access tokens

* Change "MUST" to "SHOULD" for refresh token rotation

* Editorial clarifications

-06

* Added refresh token requirements to AS summary

* Editorial clarifications

-05

* Incorporated editorial and substantive feedback from Mike Jones

* Added references to "nonce" as another way to prevent CSRF attacks

* Updated headers in the Implicit Flow section to better represent the relationship between the paragraphs

-04

* Disallow the use of the Password Grant

* Add PKCE support to summary list for authorization server requirements

* Rewrote refresh token section to allow refresh tokens if they are time-limited, rotated on each use, and requiring that the rotated refresh token lifetimes do not extend past the lifetime of the initial refresh token, and to bring it in line with the Security BCP

* Updated recommendations on using state to reflect the Security BCP

* Updated server support checklist to reflect latest changes

* Updated the same-domain JS architecture section to emphasize the architecture rather than domain

* Editorial clarifications in the section that talks about OpenID Connect ID tokens

-03

* Updated the historic note about the fragment URL clarifying that the Session History API means browsers can use the unmodified authorization code flow

* Rephrased "Authorization Code Flow" intro paragraph to better lead into the next two sections

* Softened "is likely a better decision to avoid using OAuth entirely" to "it may be..." for common-domain deployments

* Updated abstract to not be limited to public clients, since the later sections talk about confidential clients

* Removed references to avoiding OpenID Connect for same-domain architectures

* Updated headers to better describe architectures (Apps Served from a Static Web Server -> JavaScript Applications without a Backend)

* Expanded "same-domain architecture" section to better explain the problems that OAuth has in this scenario

* Referenced Security BCP in implicit flow attacks where possible

* Minor typo corrections

-02

* Rewrote overview section incorporating feedback from Leo Tohill

* Updated summary recommendation bullet points to split out application and server requirements

* Removed the allowance on hostname-only redirect URI matching, now requiring exact redirect URI matching

* Updated Section 6.2 to drop reference of SPA with a backend component being a public client

* Expanded the architecture section to explicitly mention three architectural patterns available to JS apps

-01

* Incorporated feedback from Torsten Lodderstedt

* Updated abstract

* Clarified the definition of browser-based apps to not exclude applications cached in the browser, e.g. via Service Workers

* Clarified use of the state parameter for CSRF protection

* Added background information about the original reason the implicit flow was created due to lack of CORS support

* Clarified the same-domain use case where the SPA and API share a cookie domain

* Moved historic note about the fragment URL into the Overview

Appendix C. Acknowledgements

The authors would like to acknowledge the work of William Denniss and John Bradley, whose recommendation for native apps informed many of the best practices for browser-based applications. The authors would also like to thank Hannes Tschofenig and Torsten Lodderstedt, the attendees of the Internet Identity Workshop 27 session at which this BCP was originally proposed, and the following individuals who contributed ideas, feedback, and wording that shaped and formed the final specification:

Annabelle Backman, Brian Campbell, Brock Allen, Christian Mainka, Daniel Fett, George Fletcher, Hannes Tschofenig, Janak Amarasena, John Bradley, Joseph Heenan, Justin Richer, Karl McGuinness, Karsten Meyer zu Selhausen, Leo Tohill, Mike Jones, Philippe De Ryck, Tomek Stojecki, Torsten Lodderstedt, Vittorio Bertocci and Yannick Majoros.

Authors' Addresses

Aaron Parecki
Okta
Email: aaron@parecki.com
URI: https://aaronparecki.com

David Waite
Ping Identity
Email: david@alkaline-solutions.com

Chapter 7

Draft: OAuth 2.0 Security Best Current Practice

OAuth 2.0 Security Best Current Practice describes security requirements and other recommendations for clients and servers implementing OAuth 2.0. This is a new Best Current Practice around OAuth security, intended to capture experience gained from live deployments in the years since the first Security Considerations RFC was published in 2013.

This spec describes some more advanced threats and attacks, as well as recommends against using the Implicit or Password flows entirely.

This spec is also still in draft form, so will likely go through a few more changes before it is finalized as an RFC.

Some of the concrete recommendations in this draft are deprecating the Implicit flow and Password grant, and recommending that a new refresh token is issued every time one is used.

Web Authorization Protocol
Internet-Draft
Intended status: Best Current Practice
Expires: 31 March 2023

T. Lodderstedt
yes.com
J. Bradley
Yubico
A. Labunets
Independent Researcher
D. Fett
yes.com

27 September 2022

OAuth 2.0 Security Best Current Practice
draft-ietf-oauth-security-topics-21

Abstract

This document describes best current security practice for OAuth 2.0. It updates and extends the OAuth 2.0 Security Threat Model to incorporate practical experiences gathered since OAuth 2.0 was published and covers new threats relevant due to the broader application of OAuth 2.0.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 31 March 2023.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights

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

Table of Contents

1. Introduction 3
1.1. Structure 5
1.2. Conventions and Terminology 5
2. Best Practices 5
2.1. Protecting Redirect-Based Flows 5
2.1.1. Authorization Code Grant 6
2.1.2. Implicit Grant 7
2.2. Token Replay Prevention 8
2.2.1. Access Tokens 8
2.2.2. Refresh Tokens 8
2.3. Access Token Privilege Restriction 8
2.4. Resource Owner Password Credentials Grant 9
2.5. Client Authentication 9
2.6. Other Recommendations 9
3. The Updated OAuth 2.0 Attacker Model 10
4. Attacks and Mitigations 12
4.1. Insufficient Redirect URI Validation 13
4.1.1. Redirect URI Validation Attacks on Authorization Code Grant 13
4.1.2. Redirect URI Validation Attacks on Implicit Grant . . 15
4.1.3. Countermeasures 16
4.2. Credential Leakage via Referer Headers 17
4.2.1. Leakage from the OAuth Client 17
4.2.2. Leakage from the Authorization Server 17
4.2.3. Consequences 18
4.2.4. Countermeasures 18
4.3. Credential Leakage via Browser History 19
4.3.1. Authorization Code in Browser History 19
4.3.2. Access Token in Browser History 19
4.4. Mix-Up Attacks 20
4.4.1. Attack Description 20
4.4.2. Countermeasures 22
4.5. Authorization Code Injection 23
4.5.1. Attack Description 24
4.5.2. Discussion 25
4.5.3. Countermeasures 26
4.5.4. Limitations 28
4.6. Access Token Injection 28
4.6.1. Countermeasures 28
4.7. Cross Site Request Forgery 29
4.7.1. Countermeasures 29

4.8. PKCE Downgrade Attack 30
4.8.1. Attack Description 30
4.8.2. Countermeasures 31
4.9. Access Token Leakage at the Resource Server 32
4.9.1. Access Token Phishing by Counterfeit Resource Server 32
4.9.2. Compromised Resource Server 37
4.10. Open Redirection 38
4.10.1. Client as Open Redirector 38
4.10.2. Authorization Server as Open Redirector 38
4.11. 307 Redirect 39
4.12. TLS Terminating Reverse Proxies 40
4.13. Refresh Token Protection 41
4.13.1. Discussion 41
4.13.2. Recommendations 41
4.14. Client Impersonating Resource Owner 43
4.14.1. Countermeasures 43
4.15. Clickjacking 43
4.16. Authorization Server Redirecting to Phishing Site 44
5. Acknowledgements 45
6. IANA Considerations 45
7. Security Considerations 46
8. Normative References 46
9. Informative References 47
Appendix A. Document History 51
Authors' Addresses 56

1. Introduction

Since its publication in [RFC6749] and [RFC6750], OAuth 2.0 ("OAuth" in the following) has gotten massive traction in the market and became the standard for API protection and the basis for federated login using OpenID Connect [OpenID.Core]. While OAuth is used in a variety of scenarios and different kinds of deployments, the following challenges can be observed:

* OAuth implementations are being attacked through known implementation weaknesses and anti-patterns. Although most of these threats are discussed in the OAuth 2.0 Threat Model and Security Considerations [RFC6819], continued exploitation demonstrates a need for more specific recommendations, easier to implement mitigations, and more defense in depth.

* OAuth is being used in environments with higher security requirements than considered initially, such as Open Banking, eHealth, eGovernment, and Electronic Signatures. Those use cases call for stricter guidelines and additional protection.

* OAuth is being used in much more dynamic setups than originally anticipated, creating new challenges with respect to security. Those challenges go beyond the original scope of [RFC6749], [RFC6750], and [RFC6819].

  OAuth initially assumed static relationships between client, authorization server, and resource servers. The URLs of the AS and RS were known to the client at deployment time and built an anchor for the trust relationships among those parties. The validation of whether the client talks to a legitimate server was 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 could be used to access services of different providers (in case of standard APIs, such as e-mail or OpenID Connect) or serve as a front end to a particular tenant in a multi-tenant environment. Extensions of OAuth, such as the OAuth 2.0 Dynamic Client Registration Protocol [RFC7591] and OAuth 2.0 Authorization Server Metadata [RFC8414] were developed to support the use of OAuth in dynamic scenarios.

* Technology has changed. For example, the way browsers treat fragments when redirecting requests has changed, and with it, the implicit grant's underlying security model.

This document provides updated security recommendations to address these challenges. It does not supplant the security advice given in [RFC6749], [RFC6750], and [RFC6819], but complements those documents.

This document introduces new requirements beyond those defined in existing specifications such as OAuth 2.0 [RFC6749] and OpenID Connect [OpenID.Core] and deprecates some modes of operation that are deemed less secure or even insecure. Naturally, not all existing ecosystems and implementations are compatible with the new requirements and following the best practices described in this document may break interoperability. Nonetheless, it is RECOMMENDED that implementers upgrade their implementations and ecosystems when feasible.

1.1. Structure

The remainder of this document is organized as follows: The next section summarizes the most important best practices for every OAuth implementor. Afterwards, the updated the OAuth attacker model is presented. Subsequently, a detailed analysis of the threats and implementation issues that can be found in the wild today is given along with a discussion of potential countermeasures.

1.2. Conventions and Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

This specification uses the terms "access token", "authorization endpoint", "authorization grant", "authorization server", "client", "client identifier" (client ID), "protected resource", "refresh token", "resource owner", "resource server", and "token endpoint" defined by OAuth 2.0 [RFC6749].

2. Best Practices

This section describes the set of security mechanisms and measures the OAuth working group considers best practices at the time of writing.

2.1. Protecting Redirect-Based Flows

When comparing client redirect URIs against pre-registered URIs, authorization servers MUST utilize exact string matching except for port numbers in localhost redirection URIs of native apps, see Section 4.1.3. This measure contributes to the prevention of leakage of authorization codes and access tokens (see Section 4.1). It can also help to detect mix-up attacks (see Section 4.4).

Clients and AS MUST NOT expose URLs that forward the user's browser to arbitrary URIs obtained from a query parameter ("open redirector"). Open redirectors can enable exfiltration of authorization codes and access tokens, see Section 4.10.1.

Clients MUST prevent Cross-Site Request Forgery (CSRF). In this context, CSRF refers to requests to the redirection endpoint that do not originate at the authorization server, but a malicious third party (see Section 4.4.1.8. of [RFC6819] for details). Clients that have ensured that the authorization server supports PKCE [RFC7636]

MAY rely on the CSRF protection provided by PKCE. In OpenID Connect flows, the nonce parameter provides CSRF protection. Otherwise, one-time use CSRF tokens carried in the state parameter that are securely bound to the user agent MUST be used for CSRF protection (see Section 4.7.1).

When an OAuth client can interact with more than one authorization server, a defense against mix-up attacks (see Section 4.4) is REQUIRED. To this end, clients SHOULD

* use the iss parameter as a countermeasure according to [RFC9207], or

* use an alternative countermeasure based on an iss value in the authorization response (such as the iss Claim in the ID Token in [OpenID.Core] or in [JARM] responses), processing it as described in [RFC9207].

In the absence of these options, clients MAY instead use distinct redirect URIs to identify authorization endpoints and token endpoints, as described in Section 4.4.2.

An AS that redirects a request potentially containing user credentials MUST avoid forwarding these user credentials accidentally (see Section 4.11 for details).

2.1.1. Authorization Code Grant

Clients MUST prevent authorization code injection attacks (see Section 4.5) and misuse of authorization codes using one of the following options:

* Public clients MUST use PKCE [RFC7636] to this end, as motivated in Section 4.5.3.1.

* For confidential clients, the use of PKCE [RFC7636] is RECOMMENDED, as it provides a strong protection against misuse and injection of authorization codes as described in Section 4.5.3.1 and, as a side-effect, prevents CSRF even in presence of strong attackers as described in Section 4.7.1.

* With additional precautions, described in Section 4.5.3.2, confidential OpenID Connect [OpenID.Core] clients MAY use the nonce parameter and the respective Claim in the ID Token instead.

In any case, the PKCE challenge or OpenID Connect nonce MUST be transaction-specific and securely bound to the client and the user agent in which the transaction was started.

Note: Although PKCE was designed as a mechanism to protect native apps, this advice applies to all kinds of OAuth clients, including web applications.

When using PKCE, clients SHOULD use PKCE code challenge methods that do not expose the PKCE verifier in the authorization request. Otherwise, attackers that can read the authorization request (cf. Attacker A4 in Section 3) can break the security provided by PKCE. Currently, S256 is the only such method.

Authorization servers MUST support PKCE [RFC7636].

If a client sends a valid PKCE [RFC7636] code_challenge parameter in the authorization request, the authorization server MUST enforce the correct usage of code_verifier at the token endpoint.

Authorization servers MUST mitigate PKCE Downgrade Attacks by ensuring that a token request containing a code_verifier parameter is accepted only if a code_challenge parameter was present in the authorization request, see Section 4.8.2 for details.

Authorization servers MUST provide a way to detect their support for PKCE. It is RECOMMENDED for AS to publish the element code_challenge_methods_supported in their AS metadata ([RFC8414]) containing the supported PKCE challenge methods (which can be used by the client to detect PKCE support). ASs MAY instead provide a deployment-specific way to ensure or determine PKCE support by the AS.

2.1.2. Implicit Grant

The implicit grant (response type "token") and other response types causing the authorization server to issue access tokens in the authorization response are vulnerable to access token leakage and access token replay as described in Section 4.1, Section 4.2, Section 4.3, and Section 4.6.

Moreover, no viable method for sender-constraining exists to bind access tokens to a specific client (as recommended in Section 2.2) when the access tokens are issued in the authorization response. This means that an attacker can use leaked or stolen access token at a resource endpoint.

In order to avoid these issues, clients SHOULD NOT use the implicit grant (response type "token") or other response types issuing access tokens in the authorization response, unless access token injection in the authorization response is prevented and the aforementioned token leakage vectors are mitigated.

Clients SHOULD instead use the response type "code" (aka authorization code grant type) as specified in Section 2.1.1 or any other response type that causes the authorization server to issue access tokens in the token response, such as the "code id_token" response type. This allows the authorization server to detect replay attempts by attackers and generally reduces the attack surface since access tokens are not exposed in URLs. It also allows the authorization server to sender-constrain the issued tokens (see next section).

2.2. Token Replay Prevention

2.2.1. Access Tokens

A sender-constrained access token scopes the applicability of an access token to a certain sender. This sender is obliged to demonstrate knowledge of a certain secret as prerequisite for the acceptance of that token at the recipient (e.g., a resource server).

Authorization and resource servers SHOULD use mechanisms for sender-constraining access tokens, such as Mutual TLS for OAuth 2.0 [RFC8705] or OAuth Demonstration of Proof of Possession (DPoP) [I-D.ietf-oauth-dpop] (see Section 4.9.1.1.2), to prevent misuse of stolen and leaked access tokens.

2.2.2. Refresh Tokens

Refresh tokens for public clients MUST be sender-constrained or use refresh token rotation as described in Section 4.13. [RFC6749] already mandates that refresh tokens for confidential clients can only be used by the client for which they were issued.

2.3. Access Token Privilege Restriction

The privileges associated with an access token SHOULD be restricted to the minimum required for the particular application or use case. This prevents clients from exceeding the privileges authorized by the resource owner. It also prevents users from exceeding their privileges authorized by the respective security policy. Privilege restrictions also help to reduce the impact of access token leakage.

In particular, access tokens SHOULD be restricted to certain resource servers (audience restriction), preferably to a single resource server. To put this into effect, the authorization server associates the access token with certain resource servers and every resource server is obliged to verify, for every request, whether the access token sent with that request was meant to be used for that particular resource server. If not, the resource server MUST refuse to serve

the respective request. The aud claim as defined in [RFC9068] MAY be used to audience-restrict access tokens. Clients and authorization servers MAY utilize the parameters scope or resource as specified in [RFC6749] and [RFC8707], respectively, to determine the resource server they want to access.

Additionally, access tokens SHOULD be restricted to certain resources and actions on resource servers or resources. To put this into effect, the authorization server associates the access token with the respective resource and actions and every resource server is obliged to verify, for every request, whether the access token sent with that request was meant to be used for that particular action on the particular resource. If not, the resource server must refuse to serve the respective request. Clients and authorization servers MAY utilize the parameter scope as specified in [RFC6749] and authorization_details as specified in [I-D.ietf-oauth-rar] to determine those resources and/or actions.

2.4. Resource Owner Password Credentials Grant

The resource owner password credentials grant MUST NOT be used. This grant type insecurely exposes the credentials of the resource owner to the client. Even if the client is benign, this results in an increased attack surface (credentials can leak in more places than just the AS) and users are trained to enter their credentials in places other than the AS.

Furthermore, adapting the resource owner password credentials grant to two-factor authentication, authentication with cryptographic credentials (cf. WebCrypto [WebCrypto], WebAuthn [WebAuthn]), and authentication processes that require multiple steps can be hard or impossible.

2.5. Client Authentication

Authorization servers SHOULD use client authentication if possible.

It is RECOMMENDED to use asymmetric (public-key based) methods for client authentication such as mTLS [RFC8705] or private_key_jwt [OpenID.Core]. When asymmetric methods for client authentication are used, authorization servers do not need to store sensitive symmetric keys, making these methods more robust against a number of attacks.

2.6. Other Recommendations

The use of OAuth Metadata [RFC8414] can help to improve the security of OAuth deployments:

* It ensures that security features and other new OAuth features can be enabled automatically by compliant software libraries.

* It reduces chances for misconfigurations, for example misconfigured endpoint URLs (that might belong to an attacker) or misconfigured security features.

* It can help to facilitate rotation of cryptographic keys and to ensure cryptographic agility.

It is therefore RECOMMENDED that ASs publish OAuth metadata according to [RFC8414] and that clients make use of this metadata to configure themselves when available.

Authorization servers SHOULD NOT allow clients to influence their client_id or any other Claim if that can cause confusion with a genuine resource owner, as described in Section 4.14

It is RECOMMENDED to use end-to-end TLS. If TLS traffic needs to be terminated at an intermediary, refer to Section 4.12 for further security advice.

Authorization responses MUST NOT be transmitted over unencrypted network connections. To this end, AS MUST NOT allow redirect URIs that use the http scheme except for native clients that use Loopback Interface Redirection as described in [RFC8252], Section 7.3.

Authorization servers MUST take precautions to prevent phishing attacks via redirection as described in Section 4.16.

3. The Updated OAuth 2.0 Attacker Model

In [RFC6819], an attacker model is laid out that describes the capabilities of attackers against which OAuth deployments must be protected. In the following, this attacker model is updated to account for the potentially dynamic relationships involving multiple parties (as described in Section 1), to include new types of attackers and to define the attacker model more clearly.

OAuth MUST ensure that the authorization of the resource owner (RO) (with a user agent) at the authorization server (AS) and the subsequent usage of the access token at the resource server (RS) is protected at least against the following attackers:

* (A1) Web Attackers that can set up and operate an arbitrary number of network endpoints including browsers and servers (except for the concrete RO, AS, and RS). Web attackers may set up web sites that are visited by the RO, operate their own user agents, and participate in the protocol.

  Web attackers may, in particular, operate OAuth clients that are registered at AS, and operate their own authorization and resource servers that can be used (in parallel) by the RO and other resource owners.

  It must also be assumed that web attackers can lure the user to open arbitrary attacker-chosen URIs at any time. In practice, this can be achieved in many ways, for example, by injecting malicious advertisements into advertisement networks, or by sending legitimate-looking emails.

  Web attackers can use their own user credentials to create new messages as well as any secrets they learned previously. For example, if a web attacker learns an authorization code of a user through a misconfigured redirect URI, the web attacker can then try to redeem that code for an access token.

  They cannot, however, read or manipulate messages that are not targeted towards them (e.g., sent to a URL controlled by a non-attacker controlled AS).

* (A2) Network Attackers that additionally have full control over the network over which protocol participants communicate. They can eavesdrop on, manipulate, and spoof messages, except when these are properly protected by cryptographic methods (e.g., TLS). Network attackers can also block arbitrary messages.

While an example for a web attacker would be a customer of an internet service provider, network attackers could be the internet service provider itself, an attacker in a public (wifi) network using ARP spoofing, or a state-sponsored attacker with access to internet exchange points, for instance.

These attackers conform to the attacker model that was used in formal analysis efforts for OAuth [arXiv.1601.01229]. This is a minimal attacker model. Implementers MUST take into account all possible types of attackers in the environment in which their OAuth implementations are expected to run. Previous attacks on OAuth have shown that OAuth deployments SHOULD in particular consider the following, stronger attackers in addition to those listed above:

* (A3) Attackers that can read, but not modify, the contents of the authorization response (i.e., the authorization response can leak to an attacker).

  Examples for such attacks include open redirector attacks, insufficient checking of redirect URIs (see Section 4.1), problems existing on mobile operating systems (where different apps can register themselves on the same URI), mix-up attacks (see Section 4.4), where the client is tricked into sending credentials to a attacker-controlled AS, and the fact that URLs are often stored/logged by browsers (history), proxy servers, and operating systems.

* (A4) Attackers that can read, but not modify, the contents of the authorization request (i.e., the authorization request can leak, in the same manner as above, to an attacker).

* (A5) Attackers that can acquire an access token issued by AS. For example, a resource server can be compromised by an attacker, an access token may be sent to an attacker-controlled resource server due to a misconfiguration, or an RO is social-engineered into using a attacker-controlled RS. See also Section 4.9.2.

(A3), (A4) and (A5) typically occur together with either (A1) or (A2). Attackers can collaborate to reach a common goal.

Note that in this attacker model, an attacker (see A1) can be a RO or act as one. For example, an attacker can use his own browser to replay tokens or authorization codes obtained by any of the attacks described above at the client or RS.

This document focusses on threats resulting from these attackers. Attacks in an even stronger attacker model are discussed, for example, in [arXiv.1901.11520].

4. Attacks and Mitigations

This section gives a detailed description of attacks on OAuth implementations, along with potential countermeasures. Attacks and mitigations already covered in [RFC6819] are not listed here, except where new recommendations are made.

4.1. Insufficient Redirect URI Validation

Some authorization servers allow clients to register redirect URI patterns instead of complete redirect URIs. The authorization servers then match the redirect URI parameter value at the authorization endpoint against the registered patterns at runtime. This approach allows clients to encode transaction state into additional redirect URI parameters or to register a single pattern for multiple redirect URIs.

This approach turned out to be more complex to implement and more error prone to manage than exact redirect URI matching. Several successful attacks exploiting flaws in the pattern matching implementation or concrete configurations have been observed in the wild . Insufficient validation of the redirect URI 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

* by directly sending the user agent to a URI under the attackers control, or

* 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.

These attacks are shown in detail in the following subsections.

4.1.1. Redirect URI Validation Attacks on Authorization Code Grant

For a client using the grant type code, an attack may work as follows:

Assume the redirect URL pattern https://*.somesite.example/* is registered for the client with the client ID s6BhdRkqt3. The intention is to allow any subdomain of somesite.example to be a valid redirect URI for the client, for example https://app1.somesite.example/redirect. A naive implementation on the authorization server, however, might interpret the wildcard * as "any character" and not "any character valid for a domain name". The authorization server, therefore, might permit https://attacker.example/.somesite.example as a redirect URI, although attacker.example is a different domain potentially controlled by a malicious party.

The attack can then be conducted as follows:

First, the attacker needs to trick the user into opening a tampered URL in his browser that launches a page under the attacker's control, say https://www.evil.example (see Attacker A1 in Section 3).

This URL initiates the following authorization request with the client ID of a legitimate client to the authorization endpoint (line breaks for display only):

```
GET /authorize?response_type=code&client_id=s6BhdRkqt3&state=9ad67f13
     &redirect_uri=https%3A%2F%2Fattacker.example%2F.somesite.example
     HTTP/1.1
Host: server.somesite.example
```

The authorization server validates the redirect URI and compares it to the registered redirect URL patterns for the client s6BhdRkqt3. The authorization request is processed and presented to the user.

If the user does not see the redirect URI or does not recognize the attack, the code is issued and immediately sent to the attacker's domain. If an automatic approval of the authorization is enabled (which is not recommended for public clients according to [RFC6749]), the attack can be performed even without user interaction.

If the attacker impersonated a public client, the attacker can exchange the code for tokens at the respective token endpoint.

This attack will not work as easily for confidential clients, since the code exchange requires authentication with the legitimate client's secret. The attacker can, however, use the legitimate confidential client to redeem the code by performing an authorization code injection attack, see Section 4.5.

Note: Vulnerabilities of this kind can also exist if the authorization server handles wildcards properly. For example, assume that the client registers the redirect URL pattern https://*.somesite.example/* and the authorization server interprets this as "allow redirect URIs pointing to any host residing in the domain somesite.example". If an attacker manages to establish a host or subdomain in somesite.example, he can impersonate the legitimate client. This could be caused, for example, by a subdomain takeover attack [subdomaintakeover], where an outdated CNAME record (say, external-service.somesite.example) points to an external DNS name that does no longer exist (say, customer-abc.service.example) and can be taken over by an attacker (e.g., by registering as customer-abc with the external service).

4.1.2. Redirect URI Validation 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 attack. 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 (see Section 4.10.1) in order to get access to access tokens. This allows circumvention even of very narrow redirect URI patterns, but not strict URL matching.

Assume the registered URL pattern for client s6BhdRkqt3 is https://client.somesite.example/cb?*, i.e., any parameter is allowed for redirects to https://client.somesite.example/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 an HTTP Location header redirect 303.

The attack can now be conducted as follows:

First, and as above, the attacker needs to trick the user into opening a tampered URL in his browser that launches a page under the attacker's control, say https://www.evil.example.

Afterwards, the website initiates an authorization request that is very similar to the one in the attack on the code flow. Different to above, it utilizes the open redirector by encoding redirect_to=https://attacker.example into the parameters of the redirect URI and it uses the response type "token" (line breaks for display only):

```
GET /authorize?response_type=token&state=9ad67f13
    &client_id=s6BhdRkqt3
    &redirect_uri=https%3A%2F%2Fclient.somesite.example
     %2Fcb%26redirect_to%253Dhttps%253A%252F
     %252Fattacker.example%252F HTTP/1.1
Host: server.somesite.example
```

Now, since the redirect URI matches the registered pattern, the authorization server permits the request and sends the resulting access token in a 303 redirect (some response parameters omitted for readability):

```
HTTP/1.1 303 See Other
Location: https://client.somesite.example/cb?
          redirect_to%3Dhttps%3A%2F%2Fattacker.example%2Fcb
          #access_token=2YotnFZFEjr1zCsicMWpAA&...
```

At example.com, the request arrives at the open redirector. The endpoint will read the redirect parameter and will issue an HTTP 303 Location header redirect to the URL https://attacker.example/.

```
HTTP/1.1 303 See Other
Location: https://attacker.example/
```

Since the redirector at client.somesite.example does not include a fragment in the Location header, the user agent will re-attach the original fragment #access_token=2YotnFZFEjr1zCsicMWpAA&... to the URL and will navigate to the following URL:

```
https://attacker.example/#access_token=2YotnFZFEjr1z...
```

The attacker's page at attacker.example can now access the fragment and obtain the access token.

4.1.3. Countermeasures

The complexity of implementing and managing pattern matching correctly obviously causes security issues. This document therefore advises to simplify the required logic and configuration by using exact redirect URI matching. This means the authorization server MUST compare the two URIs using simple string comparison as defined in [RFC3986], Section 6.2.1. The only exception are native apps using a localhost URI: In this case, the AS MUST allow variable port numbers as described in [RFC8252], Section 7.3.

Additional recommendations:

* Servers on which callbacks are hosted MUST NOT expose open redirectors (see Section 4.10).

* Browsers reattach URL fragments to Location redirection URLs only if the URL in the Location header does not already contain a fragment. Therefore, servers MAY prevent browsers from reattaching fragments to redirection URLs by attaching an arbitrary fragment identifier, for example #_, to URLs in Location headers.

* Clients SHOULD use the authorization code response type instead of response types causing access token issuance at the authorization endpoint. This offers countermeasures against reuse of leaked

credentials through the exchange process with the authorization server and token replay through sender-constraining of the access tokens.

If the origin and integrity of the authorization request containing the redirect URI can be verified, for example when using [RFC9101] or [RFC9126] with client authentication, the authorization server MAY trust the redirect URI without further checks.

4.2. Credential Leakage via Referer Headers

The contents of the authorization request URI or the authorization response URI can unintentionally be disclosed to attackers through the Referer HTTP header (see [RFC7231], Section 5.5.2), by leaking either from the AS's or the client's web site, respectively. Most importantly, authorization codes or state values can be disclosed in this way. Although specified otherwise in [RFC7231], Section 5.5.2, the same may happen to access tokens conveyed in URI fragments due to browser implementation issues, as illustrated by Chromium Issue 168213 [bug.chromium].

4.2.1. Leakage from the OAuth Client

Leakage from the OAuth client requires that the client, as a result of a successful authorization request, renders a page that

* contains links to other pages under the attacker's control and a user clicks on such a link, or

* includes third-party content (advertisements in iframes, images, etc.), for example if the page contains user-generated content (blog).

As soon as the browser navigates to the attacker's page or loads the third-party content, the attacker receives the authorization response URL and can extract code or state (and potentially access token).

4.2.2. Leakage from the Authorization Server

In a similar way, an attacker can learn state from the authorization request if the authorization endpoint at the authorization server contains links or third-party content as above.

4.2.3. Consequences

An attacker that learns a valid code or access token through a Referer header can perform the attacks as described in Section 4.1.1, Section 4.5, and Section 4.6. If the attacker learns state, the CSRF protection achieved by using state is lost, resulting in CSRF attacks as described in [RFC6819], Section 4.4.1.8.

4.2.4. Countermeasures

The page rendered as a result of the OAuth authorization response and the authorization endpoint SHOULD NOT include third-party resources or links to external sites.

The following measures further reduce the chances of a successful attack:

* Suppress the Referer header by applying an appropriate Referrer Policy [webappsec-referrer-policy] to the document (either as part of the "referrer" meta attribute or by setting a Referrer-Policy header). For example, the header Referrer-Policy: no-referrer in the response completely suppresses the Referer header in all requests originating from the resulting document.

* Use authorization code instead of response types causing access token issuance from the authorization endpoint.

* Bind the authorization code to a confidential client or PKCE challenge. In this case, the attacker lacks the secret to request the code exchange.

* As described in [RFC6749], Section 4.1.2, authorization codes MUST be invalidated by the AS after their first use at the token endpoint. For example, if an AS invalidated the code after the legitimate client redeemed it, the attacker would fail exchanging this code later.

  This does not mitigate the attack if the attacker manages to exchange the code for a token before the legitimate client does so. Therefore, [RFC6749] further recommends that, when an attempt is made to redeem a code twice, the AS SHOULD revoke all tokens issued previously based on that code.

* The state value SHOULD be invalidated by the client after its first use at the redirection endpoint. If this is implemented, and an attacker receives a token through the Referer header from the client's web site, the state was already used, invalidated by the client and cannot be used again by the attacker. (This does

not help if the state leaks from the AS's web site, since then the state has not been used at the redirection endpoint at the client yet.)

* Use the form post response mode instead of a redirect for the authorization response (see [OAuth.Post]).

4.3. Credential Leakage via Browser History

Authorization codes and access tokens can end up in the browser's history of visited URLs, enabling the attacks described in the following.

4.3.1. Authorization Code in Browser History

When a browser navigates to client.example/redirection_endpoint?code=abcd as a result of a redirect from a provider's authorization endpoint, the URL including the authorization code may end up in the browser's history. An attacker with access to the device could obtain the code and try to replay it.

Countermeasures:

* Authorization code replay prevention as described in [RFC6819], Section 4.4.1.1, and Section 4.5.

* Use form post response mode instead of redirect for the authorization response (see [OAuth.Post]).

4.3.2. Access Token in Browser History

An access token may end up in the browser history if a client or a web site that already has a token deliberately navigates to a page like provider.com/get_user_profile?access_token=abcdef. [RFC6750] discourages this practice and advises to transfer tokens via a header, but in practice web sites often pass access tokens in query parameters.

In case of the implicit grant, a URL like client.example/redirection_endpoint#access_token=abcdef may also end up in the browser history as a result of a redirect from a provider's authorization endpoint.

Countermeasures:

* Clients MUST NOT pass access tokens in a URI query parameter in the way described in Section 2.3 of [RFC6750]. The authorization code grant or alternative OAuth response modes like the form post response mode [OAuth.Post] can be used to this end.

4.4. Mix-Up Attacks

Mix-up is an attack on scenarios where an OAuth client interacts with two or more authorization servers and at least one authorization server is under the control of the attacker. This can be the case, for example, if the attacker uses dynamic registration to register the client at his own authorization server or if an authorization server becomes compromised.

The goal of the attack is to obtain an authorization code or an access token for an uncompromised authorization server. This is achieved by tricking the client into sending those credentials to the compromised authorization server (the attacker) instead of using them at the respective endpoint of the uncompromised authorization/ resource server.

4.4.1. Attack Description

The description here follows [arXiv.1601.01229], with variants of the attack outlined below.

Preconditions: For this variant of the attack to work, we assume that

* the implicit or authorization code grant are used with multiple AS of which one is considered "honest" (H-AS) and one is operated by the attacker (A-AS), and

* the client stores the AS chosen by the user in a session bound to the user's browser and uses the same redirection endpoint URI for each AS.

In the following, we assume that the client is registered with H-AS (URI: https://honest.as.example, client ID: 7ZGZldHQ) and with A-AS (URI: https://attacker.example, client ID: 666RVZJTA). URLs shown in the following example are shortened for presentation to only include parameters relevant for the attack.

Attack on the authorization code grant:

1. The user selects to start the grant using A-AS (e.g., by clicking on a button at the client's website).

2. The client stores in the user's session that the user selected "A-AS" and redirects the user to A-AS's authorization endpoint with a Location header containing the URL https://attacker.example/ authorize?response_type=code&client_id=666RVZJTA.

3. When the user's browser navigates to the attacker's authorization endpoint, the attacker immediately redirects the browser to the authorization endpoint of H-AS. In the authorization request, the attacker replaces the client ID of the client at A-AS with the client's ID at H-AS. Therefore, the browser receives a redirection (303 See Other) with a Location header pointing to https://honest.as.example/ authorize?response_type=code&client_id=7ZGZldHQ

4. The user authorizes the client to access her resources at H-AS. (Note that a vigilant user might at this point detect that she intended to use A-AS instead of H-AS. The first attack variant listed below avoids this.) H-AS issues a code and sends it (via the browser) back to the client.

5. Since the client still assumes that the code was issued by A-AS, it will try to redeem the code at A-AS's token endpoint.

6. The attacker therefore obtains code and can either exchange the code for an access token (for public clients) or perform an authorization code injection attack as described in Section 4.5.

Variants:

* *Mix-Up With Interception*: This variant works only if the attacker can intercept and manipulate the first request/response pair from a user's browser to the client (in which the user selects a certain AS and is then redirected by the client to that AS), as in Attacker A2 (see Section 3). This capability can, for example, be the result of a man-in-the-middle attack on the user's connection to the client. In the attack, the user starts the flow with H-AS. The attacker intercepts this request and changes the user's selection to A-AS. The rest of the attack proceeds as in Steps 2 and following above.

* *Implicit Grant*: In the implicit grant, the attacker receives an access token instead of the code; the rest of the attack works as above.

* *Per-AS Redirect URIs*: If clients use different redirect URIs for different ASs, do not store the selected AS in the user's session, and ASs do not check the redirect URIs properly, attackers can

mount an attack called "Cross-Social Network Request Forgery". These attacks have been observed in practice. Refer to [oauth_security_jcs_14] for details.

* *OpenID Connect*: There are variants that can be used to attack OpenID Connect. In these attacks, the attacker misuses features of the OpenID Connect Discovery [OpenID.Discovery] mechanism or replays access tokens or ID Tokens to conduct a mix-up attack. The attacks are described in detail in [arXiv.1704.08539], Appendix A, and [arXiv.1508.04324v2], Section 6 ("Malicious Endpoints Attacks").

4.4.2. Countermeasures

When an OAuth client can only interact with one authorization server, a mix-up defense is not required. In scenarios where an OAuth client interacts with two or more authorization servers, however, clients MUST prevent mix-up attacks. Two different methods are discussed in the following.

For both defenses, clients MUST store, for each authorization request, the issuer they sent the authorization request to and bind this information to the user agent. The issuer serves, via the associated metadata, as an abstract identifier for the combination of the authorization endpoint and token endpoint that are to be used in the flow. If an issuer identifier is not available, for example, if neither OAuth metadata [RFC8414] nor OpenID Connect Discovery [OpenID.Discovery] are used, a different unique identifier for this tuple or the tuple itself can be used instead. For brevity of presentation, such a deployment-specific identifier will be subsumed under the issuer (or issuer identifier) in the following.

Note: Just storing the authorization server URL is not sufficient to identify mix-up attacks. An attacker might declare an uncompromised AS's authorization endpoint URL as "his" AS URL, but declare a token endpoint under his own control.

4.4.2.1. Mix-Up Defense via Issuer Identification

This defense requires that the authorization server sends his issuer identifier in the authorization response to the client. When receiving the authorization response, the client MUST compare the received issuer identifier to the stored issuer identifier. If there is a mismatch, the client MUST abort the interaction.

There are different ways this issuer identifier can be transported to the client:

* The issuer information can be transported, for example, via a separate response parameter iss, defined in [RFC9207].

* When OpenID Connect is used and an ID Token is returned in the authorization response, the client can evaluate the iss Claim in the ID Token.

In both cases, the iss value MUST be evaluated according to [RFC9207].

While this defense may require deploying new OAuth features to transport the issuer information, it is a robust and relatively simple defense against mix-up.

4.4.2.2. Mix-Up Defense via Distinct Redirect URIs

For this defense, clients MUST use a distinct redirect URI for each issuer they interact with.

Clients MUST check that the authorization response was received from the correct issuer by comparing the distinct redirect URI for the issuer to the URI where the authorization response was received on. If there is a mismatch, the client MUST abort the flow.

While this defense builds upon existing OAuth functionality, it cannot be used in scenarios where clients only register once for the use of many different issuers (as in some open banking schemes) and due to the tight integration with the client registration, it is harder to deploy automatically.

Furthermore, an attacker might be able to circumvent the protection offered by this defense by registering a new client with the "honest" AS using the redirect URI that the client assigned to the attacker's AS. The attacker could then run the attack as described above, replacing the client ID with the client ID of his newly created client.

This defense SHOULD therefore only be used if other options are not available.

4.5. Authorization Code Injection

An attacker that has gained access to an authorization code contained in an authorization response (see Attacker A3 in Section 3) can try to redeem the authorization code for an access token or otherwise make use of the authorization code.

In the case that the authorization code was created for a public client, the attacker can send the authorization code to the token endpoint of the authorization server and thereby get an access token. This attack was described in Section 4.4.1.1 of [RFC6819].

For confidential clients, or in some special situations, the attacker can execute an authorization code injection attack, as described in the following.

In an authorization code injection attack, the attacker attempts to inject a stolen authorization code into the attacker's own session with the client. The aim is to associate the attacker's session at the client with the victim's resources or identity, thereby giving the attacker at least limited access to the victum's resources.

Besides circumventing the client authentication of confidential clients, other use cases for this attack include:

* The attacker wants to access certain functions in this particular client. As an example, the attacker wants to impersonate his victim in a certain app or on a certain web site.

* The authorization or resource servers are limited to certain networks that the attacker is unable to access directly.

Except in these special cases, authorization code injection is usually not interesting when the code was created for a public client, as sending the code to the token endpoint is a simpler and more powerful attack, as described above.

4.5.1. Attack Description

The authorization code injection attack works as follows:

1. The attacker obtains an authorization code (see attacker A3 in Section 3). For the rest of the attack, only the capabilities of a web attacker (A1) are required.

2. From the attacker's own device, the attacker starts a regular OAuth authorization process with the legitimate client.

3. In the response of the authorization server to the legitimate client, the attacker replaces the newly created authorization code with the stolen authorization code. Since this response is passing through the attacker's device, the attacker can use any tool that can intercept and manipulate the authorization response to this end. The attacker does not need to control the network.

4. The legitimate client sends the code to the authorization server's token endpoint, along with the redirect_uri and the client's client ID and client secret (or other means of client authentication).

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 (see [RFC6749]).

6. All checks succeed and the authorization server issues access and other tokens to the client. The attacker has now associated his session with the legitimate client with the victim's resources and/or identity.

4.5.2. Discussion

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. The check will also fail if the authorization code was already redeemed by the legitimate user and was one-time use only.

An attempt to inject a code obtained via a manipulated redirect URI 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 attempts to inject an authorization code that had been obtained from another instance of the same client on another device, if certain conditions are fulfilled:

* the redirect URI itself needs to contain a nonce or another kind of one-time use, secret data and

* the client has bound this data to this particular instance of the client.

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 specification.

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 specification, since the tampered redirect URI is not considered. So any attempt to inject an authorization code obtained using the client_id of a legitimate client or by utilizing the legitimate client on another device will not 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 store or re-construct the correct redirect URI for the call to the token endpoint.

Asymmetric methods for client authentication do not stop this attack, as the legitimate client authenticates at the token endpoint.

This document therefore recommends to instead bind every authorization code to a certain client instance on a certain device (or in a certain user agent) in the context of a certain transaction using one of the mechanisms described next.

4.5.3. Countermeasures

There are two good technical solutions to achieve this goal, outlined in the following.

4.5.3.1. PKCE

The PKCE mechanism specified in [RFC7636] can be used as a countermeasure. When the attacker attempts to inject an authorization code, the check of the code_verifier fails: the client uses its correct verifier, but the code is associated with a code_challenge that does not match this verifier. PKCE is a deployed OAuth feature, although its originally intended use was solely focused on securing native apps, not the broader use recommended by this document.

PKCE does not only protect against the autorization code injection attack, but also protects authorization codes created for public clients: PKCE ensures that an attacker cannot redeem a stolen authorization code at the token endpoint of the authorization server without knowledge of the code_verifier.

4.5.3.2. Nonce

OpenID Connect's existing nonce parameter can protect against authorization code injection attacks. 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 puts the received nonce value into the ID Token that 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. The assumption is that an attacker cannot get hold of the user agent state on the victim's device, where the attacker has stolen the respective authorization code.

It is important to note that this countermeasure only works if the client properly checks the nonce parameter in the ID Token and does not use any issued token until this check has succeeded. More precisely, a client protecting itself against code injection using the nonce parameter,

1. MUST validate the nonce in the ID Token obtained from the token endpoint, even if another ID Token was obtained from the authorization response (e.g., response_type=code+id_token), and

2. MUST ensure that, unless and until that check succeeds, all tokens (ID Tokens and the access token) are disregarded and not used for any other purpose.

It is important to note that nonce does not protect authorization codes of public clients, as an attacker does not need to execute an authorization code injection attack. Instead, an attacker can directly call the token endpoint with the stolen authorization code.

4.5.3.3. Other Solutions

Other solutions, like binding state to the code, sender-constraining the code using cryptographic means, or per-instance client credentials are conceivable, but lack support and bring new security requirements.

PKCE is the most obvious solution for OAuth clients as it is available today (originally intended for OAuth native apps) whereas nonce is appropriate for OpenID Connect clients.

4.5.4. Limitations

An attacker can circumvent the countermeasures described above if he can modify the nonce or code_challenge values that are used in the victim's authorization request. The attacker can modify these values to be the same ones as those chosen by the client in his own session in Step 2 of the attack above. (This requires that the victim's session with the client begins after the attacker started his session with the client.) If the attacker is then able to capture the authorization code from the victim, the attacker will be able to inject the stolen code in Step 3 even if PKCE or nonce are used.

This attack is complex and requires a close interaction between the attacker and the victim's session. Nonetheless, measures to prevent attackers from reading the contents of the authorization response still need to be taken, as described in Section 4.1, Section 4.2, Section 4.3, Section 4.4, and Section 4.10.

4.6. Access Token Injection

In an access token injection attack, the attacker attempts to inject a stolen access token into a legitimate client (that is not under the attacker's control). This will typically happen if the attacker wants to utilize a leaked access token to impersonate a user in a certain client.

To conduct the attack, the attacker starts an OAuth flow with the client using the implicit grant and modifies the authorization response by replacing the access token issued by the authorization server or directly makes up an authorization server response including the leaked access token. Since the response includes the state value generated by the client for this particular transaction, the client does not treat the response as a CSRF attack and uses the access token injected by the attacker.

4.6.1. Countermeasures

There is no way to detect such an injection attack in pure-OAuth flows, since the token is issued without any binding to the transaction or the particular user agent.

In OpenID Connect, the attack can be mitigated, as the authorization response additionally contains an ID Token containing the at_hash claim. The attacker therefore needs to replace both the access token

as well as the ID Token in the response. The attacker cannot forge the ID Token, as it is signed or encrypted with authentication. The attacker also cannot inject a leaked ID Token matching the stolen access token, as the nonce claim in the leaked ID Token will (with a very high probability) contain a different value than the one expected in the authorization response.

Note that further protection, like sender-constrained access tokens, is still required to prevent attackers from using the access token at the resource endpoint directly.

The recommendations in Section 2.1.2 follow from this.

4.7. Cross Site Request Forgery

An attacker might attempt to inject a request to the redirect URI of the legitimate client on the victim's device, e.g., to cause the client to access resources under the attacker's control. This is a variant of an attack known as Cross-Site Request Forgery (CSRF).

4.7.1. Countermeasures

The traditional countermeasure is that clients pass a value in the state parameter that links the request to the redirect URI to the user agent session as described in detail in [RFC6819], Section 5.3.5. The same protection is provided by PKCE or the OpenID Connect nonce value.

When using PKCE instead of state or nonce for CSRF protection, it is important to note that:

* Clients MUST ensure that the AS supports PKCE before using PKCE for CSRF protection. If an authorization server does not support PKCE, state or nonce MUST be used for CSRF protection.

* If state is used for carrying application state, and integrity of its contents is a concern, clients MUST protect state against tampering and swapping. This can be achieved by binding the contents of state to the browser session and/or signed/encrypted state values as discussed in the now-expired draft [I-D.bradley-oauth-jwt-encoded-state].

The AS therefore MUST provide a way to detect their support for PKCE. Using AS metadata according to [RFC8414] is RECOMMENDED, but AS MAY instead provide a deployment-specific way to ensure or determine PKCE support.

PKCE provides robust protection against CSRF attacks even in presence of an that can read the authorization response (see Attacker A3 in Section 3). When state is used or an ID Token is returned in the authorization response (e.g., response_type=code+id_token), the attacker either learns the state value and can replay it into the forged authorization response, or can extract the nonce from the ID Token and use it in a new request to the authorization server to mint an ID Token with the same nonce. The new ID Token can then be used for the CSRF attack.

4.8. PKCE Downgrade Attack

An authorization server that supports PKCE but does not make its use mandatory for all flows can be susceptible to a PKCE downgrade attack.

The first prerequisite for this attack is that there is an attacker-controllable flag in the authorization request that enables or disables PKCE for the particular flow. The presence or absence of the code_challenge parameter lends itself for this purpose, i.e., the AS enables and enforces PKCE if this parameter is present in the authorization request, but does not enforce PKCE if the parameter is missing.

The second prerequisite for this attack is that the client is not using state at all (e.g., because the client relies on PKCE for CSRF prevention) or that the client is not checking state correctly.

Roughly speaking, this attack is a variant of a CSRF attack. The attacker achieves the same goal as in the attack described in Section 4.7: The attacker injects an authorization code (and with that, an access token) that is bound to the attacker's resources into a session between his victim and the client.

4.8.1. Attack Description

1. The user has started an OAuth session using some client at an AS. In the authorization request, the client has set the parameter code_challenge=sha256(abc) as the PKCE code challenge. The client is now waiting to receive the authorization response from the user's browse.

2. To conduct the attack, the attacker uses his own device to start an authorization flow with the targeted client. The client now uses another PKCE code challenge, say code_challenge=sha256(xyz), in the authorization request. The attacker intercepts the request and removes the entire code_challenge parameter from the request. Since this step is performed on the attacker's device, the attacker has full access to the request contents, for example using browser debug tools.

3. If the authorization server allows for flows without PKCE, it will create a code that is not bound to any PKCE code challenge.

4. The attacker now redirects the user's browser to an authorization response URL that contains the code for the attacker's session with the AS.

5. The user's browser sends the authorization code to the client, which will now try to redeem the code for an access token at the AS. The client will send code_verifier=abc as the PKCE code verifier in the token request.

6. Since the authorization server sees that this code is not bound to any PKCE code challenge, it will not check the presence or contents of the code_verifier parameter. It will issue an access token that belongs to the attacker's resource to the client under the user's control.

4.8.2. Countermeasures

Using state properly would prevent this attack. However, practice has shown that many OAuth clients do not use or check state properly.

Therefore, ASs MUST take precautions against this threat.

Note that from the view of the AS, in the attack described above, a code_verifier parameter is received at the token endpoint although no code_challenge parameter was present in the authorization request for the OAuth flow in which the authorization code was issued.

This fact can be used to mitigate this attack. [RFC7636] already mandates that

* an AS that supports PKCE MUST check whether a code challenge is contained in the authorization request and bind this information to the code that is issued; and

* when a code arrives at the token endpoint, and there was a code_challenge in the authorization request for which this code was issued, there must be a valid code_verifier in the token request.

Beyond this, to prevent PKCE downgrade attacks, the AS MUST ensure that if there was no code_challenge in the authorization request, a request to the token endpoint containing a code_verifier is rejected.

Note: ASs that mandate the use of PKCE in general or for particular clients implicitly implement this security measure.

4.9. Access Token Leakage at the Resource Server

Access tokens can leak from a resource server under certain circumstances.

4.9.1. Access Token Phishing by Counterfeit Resource Server

An attacker may setup his own resource server and trick a client into sending access tokens to it that are valid for other resource servers (see Attackers A1 and A5 in Section 3). If the client sends a valid access token to this counterfeit resource server, the attacker in turn may use that token to access other services on behalf of the resource owner.

This attack assumes the client is not bound to one specific resource server (and its URL) at development time, but client instances are provided with the resource server URL at runtime. This kind of late binding is typical in situations where the client uses a service implementing a standardized API (e.g., for e-Mail, calendar, health, or banking) and where the client is configured by a user or administrator for a service that this user or company uses.

4.9.1.1. Countermeasures

There are several potential mitigation strategies, which will be discussed in the following sections.

4.9.1.1.1. Metadata

An authorization server could provide the client with additional information about the locations where it is safe to use its access tokens.

In the simplest form, this would require the AS to publish a list of its known resource servers, illustrated in the following example using a non-standard metadata parameter resource_servers:

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

{
  "issuer":"https://server.somesite.example",
  "authorization_endpoint":
    "https://server.somesite.example/authorize",
  "resource_servers":[
    "email.somesite.example",
    "storage.somesite.example",
    "video.somesite.example"
  ]
  ...
}
```

The AS could also return the URL(s) an access token is good for in the token response, illustrated by the example and non-standard return parameter access_token_resource_server:

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "access_token":"2YotnFZFEjr1zCsicMWpAA",
  "access_token_resource_server":
    "https://hostedresource.somesite.example/path1",
...
}
```

This mitigation strategy would rely on the client to enforce the security policy and to only send access tokens to legitimate destinations. Results of OAuth-related security research (see for example [oauth_security_ubc] and [oauth_security_cmu]) indicate a large portion of client implementations do not or fail to properly implement security controls, like state checks. So relying on clients to prevent access token phishing is likely to fail as well. Moreover, given the ratio of clients to authorization and resource servers, it is considered the more viable approach to move as much as possible security-related logic to those entities. Clearly, the client has to contribute to the overall security. But there are alternative countermeasures, as described in the next sections, that provide a better balance between the involved parties.

4.9.1.1.2. Sender-Constrained Access Tokens

As the name suggests, sender-constrained access tokens scope the applicability of an access token to a certain sender. This sender is obliged to demonstrate knowledge of a certain secret as prerequisite for the acceptance of that token at a resource server.

A typical flow looks like this:

1. The authorization server associates data with the access token that binds this particular token to a certain client. The binding can utilize the client identity, but in most cases the AS utilizes key material (or data derived from the key material) known to the client.

2. This key material must be distributed somehow. Either the key material already exists before the AS creates the binding or the AS creates ephemeral keys. The way pre-existing key material is distributed varies among the different approaches. For example, X.509 Certificates can be used, in which case the distribution happens explicitly during the enrollment process. Or the key material is created and distributed at the TLS layer, in which case it might automatically happen during the setup of a TLS connection.

3. The RS must implement the actual proof of possession check. This is typically done on the application level, often tied to specific material provided by transport layer (e.g., TLS). The RS must also ensure that replay of the proof of possession is not possible.

Two methods for sender-constrained access tokens using proof-of-possession have been defined by the OAuth working group:

* *OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens* ([RFC8705]): The approach as specified in this document allows the use of mutual TLS (mTLS) for both client authentication and sender-constrained access tokens. For the purpose of sender-constrained access tokens, the client is identified towards the resource server by the fingerprint of its public key. During processing of an access token request, the authorization server obtains the client's public key from the TLS stack and associates its fingerprint with the respective access tokens. The resource server in the same way obtains the public key from the TLS stack and compares its fingerprint with the fingerprint associated with the access token.

* *DPoP* ([I-D.ietf-oauth-dpop]): DPoP (Demonstration of Proof-of-Possession at the Application Layer) outlines an application-level sender-constraining for access and refresh tokens that can be used in cases where neither mTLS nor OAuth Token Binding (see below) are available. It uses proof-of-possession based on a public/private key pair and application-level signing. DPoP can be used with public clients and, in case of confidential clients, can be combined with any client authentication method.

For reference, other approaches have been discussed as well but the relevant drafts are now expired:

* *OAuth Token Binding* ([I-D.ietf-oauth-token-binding]): In this approach, an access token is, via the token binding ID, bound to key material representing a long term association between a client and a certain TLS host. Negotiation of the key material and proof of possession in the context of a TLS handshake is taken care of by the TLS stack. The client needs to determine the token binding ID of the target resource server and pass this data to the access token request. The authorization server then associates the access token with this ID. The resource server checks on every invocation that the token binding ID of the active TLS connection and the token binding ID of associated with the access token match. Since all crypto-related functions are covered by the TLS stack, this approach is very client developer friendly. As a prerequisite, token binding as described in [RFC8473] (including federated token bindings) must be supported on all ends (client, authorization server, resource server).

* *Signed HTTP Requests* ([I-D.ietf-oauth-signed-http-request]): This approach utilizes [I-D.ietf-oauth-pop-key-distribution] and represents the elements of the signature in a JSON object. The signature is built using JWS. The mechanism has built-in support for signing of HTTP method, query parameters and headers. It also incorporates a timestamp as basis for replay prevention.

* *JWT Pop Tokens* ([I-D.sakimura-oauth-jpop]): This draft describes different ways to constrain access token usage, namely TLS or request signing. Note: Since the authors of this draft contributed the TLS-related proposal to [RFC8705], this document only considers the request signing part. For request signing, the draft utilizes [I-D.ietf-oauth-pop-key-distribution] and [RFC7800]. The signature data is represented in a JWT and JWS is used for signing. Replay prevention is provided by building the signature over a server-provided nonce, client-provided nonce and a nonce counter.

At the time of writing, OAuth Mutual TLS is the most widely implemented and the only standardized sender-constraining method.

Note that the security of sender-constrained tokens is undermined when an attacker gets access to the token and the key material. This is, in particular, the case for corrupted client software and cross-site scripting attacks (when the client is running in the browser). If the key material is protected in a hardware or software security module or only indirectly accessible (like in a TLS stack), sender-constrained tokens at least protect against a use of the token when the client is offline, i.e., when the security module or interface is not available to the attacker. This applies to access tokens as well as to refresh tokens (see Section 4.13).

4.9.1.1.3. Audience Restricted Access Tokens

Audience restriction essentially restricts access tokens to a particular resource server. The authorization server associates the access token with the particular resource server and the resource server SHOULD verify the intended audience. If the access token fails the intended audience validation, the resource server MUST refuse to serve the respective request.

In general, audience restrictions limit the impact of token leakage. In the case of a counterfeit resource server, it may (as described below) also prevent abuse of the phished access token at the legitimate resource server.

The audience can be expressed using logical names or physical addresses (like URLs). To prevent phishing, it is necessary to use the actual URL the client will send requests to. In the phishing case, this URL will point to the counterfeit resource server. If the attacker tries to use the access token at the legitimate resource server (which has a different URL), the resource server will detect the mismatch (wrong audience) and refuse to serve the request.

In deployments where the authorization server knows the URLs of all resource servers, the authorization server may just refuse to issue access tokens for unknown resource server URLs.

The client SHOULD tell the authorization server the intended resource server. The proposed mechanism [RFC8707] could be used or by encoding the information in the scope value.

Instead of the URL, it is also possible to utilize the fingerprint of the resource server's X.509 certificate as audience value. This variant would also allow to detect an attempt to spoof the legitimate resource server's URL by using a valid TLS certificate obtained from a different CA. It might also be considered a privacy benefit to hide the resource server URL from the authorization server.

Audience restriction may seem easier to use since it does not require any crypto on the client side. Still, since every access token is bound to a specific resource server, the client also needs to obtain a single RS-specific access token when accessing several resource servers. (Resource indicators, as specified in [RFC8707], can help to achieve this.) [I-D.ietf-oauth-token-binding] has the same property since different token binding IDs must be associated with the access token. Using [RFC8705], on the other hand, allows a client to use the access token at multiple resource servers.

It should be noted that audience restrictions, or generally speaking an indication by the client to the authorization server where it wants to use the access token, has additional benefits beyond the scope of token leakage prevention. It allows the authorization server to create a different access token whose format and content is specifically minted for the respective server. This has huge functional and privacy advantages in deployments using structured access tokens.

4.9.2. Compromised Resource Server

An attacker may compromise a resource server to gain access to the resources of the respective deployment. Such a compromise may range from partial access to the system, e.g., its log files, to full control of the respective server.

If the attacker were able to gain full control, including shell access, all controls can be circumvented and all resources can be accessed. The attacker would also be able to obtain other access tokens held on the compromised system that would potentially be valid to access other resource servers.

Preventing server breaches by hardening and monitoring server systems is considered a standard operational procedure and, therefore, out of the scope of this document. This section focuses on the impact of OAuth-related breaches and the replaying of captured access tokens.

The following measures should be taken into account by implementers in order to cope with access token replay by malicious actors:

* Sender-constrained access tokens, as described in Section 4.9.1.1.2, SHOULD be used to prevent the attacker from replaying the access tokens on other resource servers. Depending on the severity of the penetration, sender-constrained access tokens will also prevent replay on the compromised system.

* Audience restriction as described in Section 4.9.1.1.3 SHOULD be used to prevent replay of captured access tokens on other resource servers.

* The resource server MUST treat access tokens like any other credentials. It is considered good practice to not log them and not store them in plain text.

The first and second recommendation also apply to other scenarios where access tokens leak (see Attacker A5 in Section 3).

4.10. Open Redirection

The following attacks can occur when an AS or client has an open redirector. An open redirector is an endpoint that forwards a user's browser to an arbitrary URI obtained from a query parameter. Such endpoints are sometimes implemented, for example, to show a message before a user is then redirected to an external website, or to redirect users back to a URL they were intending to visit before being interrupted, e.g., by a login prompt.

4.10.1. Client as Open Redirector

Clients MUST NOT expose open redirectors. Attackers may use open redirectors to produce URLs pointing to the client and utilize them to exfiltrate authorization codes and access tokens, as described in Section 4.1.2. Another abuse case is to produce URLs that appear to point to the client. This might trick users into trusting the URL and follow it in their browser. This can be abused for phishing.

In order to prevent open redirection, clients should only redirect if the target URLs are whitelisted or if the origin and integrity of a request can be authenticated. Countermeasures against open redirection are described by OWASP [owasp_redir].

4.10.2. Authorization Server as Open Redirector

Just as with clients, attackers could try to utilize a user's trust in the authorization server (and its URL in particular) for performing phishing attacks. OAuth authorization servers regularly redirect users to other web sites (the clients), but must do so in a safe way.

[RFC6749], Section 4.1.2.1, already prevents open redirects by stating that the AS MUST NOT automatically redirect the user agent in case of an invalid combination of client_id and redirect_uri.

However, an attacker could also utilize a correctly registered redirect URI to perform phishing attacks. The attacker could, for example, register a client via dynamic client registration [RFC7591] and intentionally send an erroneous authorization request, e.g., by using an invalid scope value, thus instructing the AS to redirect the user agent to its phishing site.

The AS MUST take precautions to prevent this threat. Based on its risk assessment, the AS needs to decide whether it can trust the redirect URI and SHOULD only automatically redirect the user agent if it trusts the redirect URI. If the URI is not trusted, the AS MAY inform the user and rely on the user to make the correct decision.

4.11. 307 Redirect

At the authorization endpoint, a typical protocol flow is that the AS prompts the user to enter her credentials in a form that is then submitted (using the HTTP POST method) back to the authorization server. The AS checks the credentials and, if successful, redirects the user agent to the client's redirection endpoint.

In [RFC6749], the HTTP status code 302 is used for this purpose, but "any other method available via the user-agent to accomplish this redirection is allowed". When the status code 307 is used for redirection instead, the user agent will send the user's credentials via HTTP POST to the client.

This discloses the sensitive credentials to the client. If the client is malicious, it can use the credentials to impersonate the user at the AS.

The behavior might be unexpected for developers, but is defined in [RFC7231], Section 6.4.7. This status code does not require the user agent to rewrite the POST request to a GET request and thereby drop the form data in the POST request body.

In the HTTP standard [RFC7231], only the status code 303 unambigiously enforces rewriting the HTTP POST request to an HTTP GET request. For all other status codes, including the popular 302, user agents can opt not to rewrite POST to GET requests and therefore to reveal the user's credentials to the client. (In practice, however, most user agents will only show this behaviour for 307 redirects.)

ASs that redirect a request that potentially contains the user's credentials therefore MUST NOT use the HTTP 307 status code for redirection. If an HTTP redirection (and not, for example, JavaScript) is used for such a request, the AS SHOULD use HTTP status code 303 (See Other).

4.12. TLS Terminating Reverse Proxies

A common deployment architecture for HTTP applications is to hide the application server behind a reverse proxy that terminates the TLS connection and dispatches the incoming requests to the respective application server nodes.

This section highlights some attack angles of this deployment architecture with relevance to OAuth and gives recommendations for security controls.

In some situations, the reverse proxy needs to pass security-related data to the upstream application servers for further processing. Examples include the IP address of the request originator, token binding ids, and authenticated TLS client certificates. This data is usually passed in custom HTTP headers added to the upstream request.

If the reverse proxy would pass through any header sent from the outside, an attacker could try to directly send the faked header values through the proxy to the application server in order to circumvent security controls that way. For example, it is standard practice of reverse proxies to accept X-Forwarded-For headers and just add the origin of the inbound request (making it a list). Depending on the logic performed in the application server, the attacker could simply add a whitelisted IP address to the header and render a IP whitelist useless.

A reverse proxy MUST therefore sanitize any inbound requests to ensure the authenticity and integrity of all header values relevant for the security of the application servers.

If an attacker were able to get access to the internal network between proxy and application server, the attacker could also try to circumvent security controls in place. It is, therefore, essential to ensure the authenticity of the communicating entities. Furthermore, the communication link between reverse proxy and application server MUST be protected against eavesdropping, injection, and replay of messages.

4.13. Refresh Token Protection

Refresh tokens are a convenient and user-friendly way to obtain new access tokens after the expiration of access tokens. Refresh tokens also add to the security of OAuth, since they allow the authorization server to issue access tokens with a short lifetime and reduced scope, thus reducing the potential impact of access token leakage.

4.13.1. Discussion

Refresh tokens are an attractive target for attackers, since they represent the overall grant a resource owner delegated to a certain client. If an attacker is able to exfiltrate and successfully replay a refresh token, the attacker will be able to mint access tokens and use them to access resource servers on behalf of the resource owner.

[RFC6749] already provides a robust baseline protection by requiring

* confidentiality of the refresh tokens in transit and storage,

* the transmission of refresh tokens over TLS-protected connections between authorization server and client,

* the authorization server to maintain and check the binding of a refresh token to a certain client and authentication of this client during token refresh, if possible, and

* that refresh tokens cannot be generated, modified, or guessed.

[RFC6749] also lays the foundation for further (implementation specific) security measures, such as refresh token expiration and revocation as well as refresh token rotation by defining respective error codes and response behaviors.

This specification gives recommendations beyond the scope of [RFC6749] and clarifications.

4.13.2. Recommendations

Authorization servers SHOULD determine, based on a risk assessment, whether to issue refresh tokens to a certain client. If the authorization server decides not to issue refresh tokens, the client MAY refresh access tokens by utilizing other grant types, such as the authorization code grant type. In such a case, the authorization server may utilize cookies and persistent grants to optimize the user experience.

If refresh tokens are issued, those refresh tokens MUST be bound to the scope and resource servers as consented by the resource owner. This is to prevent privilege escalation by the legitimate client and reduce the impact of refresh token leakage.

For confidential clients, [RFC6749] already requires that refresh tokens can only be used by the client for which they were issued.

Authorization server MUST utilize one of these methods to detect refresh token replay by malicious actors for public clients:

* *Sender-constrained refresh tokens:* the authorization server cryptographically binds the refresh token to a certain client instance, e.g., by utilizing [RFC8705] or [I-D.ietf-oauth-dpop].

* *Refresh token rotation:* the authorization server issues a new refresh token with every access token refresh response. The previous refresh token is invalidated but information about the relationship is retained by the authorization server. If a refresh token is compromised and subsequently used by both the attacker and the legitimate client, one of them will present an invalidated refresh token, which will inform the authorization server of the breach. The authorization server cannot determine which party submitted the invalid refresh token, but it will revoke the active refresh token. This stops the attack at the cost of forcing the legitimate client to obtain a fresh authorization grant.

  Implementation note: the grant to which a refresh token belongs may be encoded into the refresh token itself. This can enable an authorization server to efficiently determine the grant to which a refresh token belongs, and by extension, all refresh tokens that need to be revoked. Authorization servers MUST ensure the integrity of the refresh token value in this case, for example, using signatures.

Authorization servers MAY revoke refresh tokens automatically in case of a security event, such as:

* password change

* logout at the authorization server

Refresh tokens SHOULD expire if the client has been inactive for some time, i.e., the refresh token has not been used to obtain fresh access tokens for some time. The expiration time is at the discretion of the authorization server. It might be a global value or determined based on the client policy or the grant associated with the refresh token (and its sensitivity).

4.14. Client Impersonating Resource Owner

Resource servers may make access control decisions based on the identity of a resource owner, for which an access token was issued, or based on the identity of a client in the client credentials grant. If both options are possible, depending on the details of the implementation, a client's identity may be mistaken for the identity of a resource owner. For example, if a client is able to choose its own client_id during registration with the authorization server, a malicious client may set it to a value identifying an end-user (e.g., a sub value if OpenID Connect is used). If the resource server cannot properly distinguish between access tokens issued to clients and access tokens issued to end-users, the client may then be able to access resource of the end-user.

4.14.1. Countermeasures

Authorization servers SHOULD NOT allow clients to influence their client_id or any other Claim if that can cause confusion with a genuine resource owner. Where this cannot be avoided, authorization servers MUST provide other means for the resource server to distinguish between access tokens authorized by a resource owner from access tokens authorized by the client itself.

4.15. Clickjacking

As described in Section 4.4.1.9 of [RFC6819], the authorization request is susceptible to clickjacking attacks, also called user interface redressing. In such an attack, an attacker embeds the authorization endpoint user interface in an innocuous context. A user believing to interact with that context, for example, clicking on buttons, inadvertently interacts with the authorization endpoint user interface instead. The opposite can be achieved as well: A user believing to interact with the authorization endpoint might inadvertently type a password into an attacker-provided input field overlaid over the original user interface. Clickjacking attacks can be designed such that users can hardly notice the attack, for example using almost invisible iframes overlaid on top of other elements.

An attacker can use this vector to obtain the user's authentication credentials, change the scope of access granted to the client, and potentially access the user's resources.

Authorization servers MUST prevent clickjacking attacks. Multiple countermeasures are described in [RFC6819], including the use of the X-Frame-Options HTTP response header field and frame-busting JavaScript. In addition to those, authorization servers SHOULD also use Content Security Policy (CSP) level 2 [CSP-2] or greater.

To be effective, CSP must be used on the authorization endpoint and, if applicable, other endpoints used to authenticate the user and authorize the client (e.g., the device authorization endpoint, login pages, error pages, etc.). This prevents framing by unauthorized origins in user agents that support CSP. The client MAY permit being framed by some other origin than the one used in its redirection endpoint. For this reason, authorization servers SHOULD allow administrators to configure allowed origins for particular clients and/or for clients to register these dynamically.

Using CSP allows authorization servers to specify multiple origins in a single response header field and to constrain these using flexible patterns (see [CSP-2] for details). Level 2 of this standard provides a robust mechanism for protecting against clickjacking by using policies that restrict the origin of frames (using frame-ancestors) together with those that restrict the sources of scripts allowed to execute on an HTML page (by using script-src). A non-normative example of such a policy is shown in the following listing:

```
HTTP/1.1 200 OK
Content-Security-Policy: frame-ancestors https://ext.example.org:8000
Content-Security-Policy: script-src 'self'
X-Frame-Options: ALLOW-FROM https://ext.example.org:8000
...
```

Because some user agents do not support [CSP-2], this technique SHOULD be combined with others, including those described in [RFC6819], unless such legacy user agents are explicitly unsupported by the authorization server. Even in such cases, additional countermeasures SHOULD still be employed.

4.16. Authorization Server Redirecting to Phishing Site

However, an attacker could also utilize a correctly registered redirect URI to perform phishing attacks. The attacker could, for example, register a client via dynamic client registration [RFC7591] and execute one of the following attacks:

1. Intentionally send an erroneous authorization request, e.g., by using an invalid scope value, thus instructing the AS to redirect the user-agent to its phishing site.

2. Intentionally send a valid authorization request with client_id and redirect_uri controlled by the attacker. After the user authenticates, the AS prompts the user to provide consent to the request. If the user notices an issue with the request and declines the request, the AS still redirects the user agent to the phishing site. In this case, the user agent will be redirected to the phishing site regardless of the action taken by the user.

3. Intentionally send a valid silent authentication request (prompt=none) with client_id and redirect_uri controlled by the attacker. In this case, the AS will automatically redirect the user agent to the phishing site.

The AS MUST take precautions to prevent these threats. The AS MUST always authenticate the user first and, with the exception of the silent authentication use case, prompt the user for credentials when needed, before redirecting the user. Based on its risk assessment, the AS needs to decide whether it can trust the redirect URI or not. It could take into account URI analytics done internally or through some external service to evaluate the credibility and trustworthiness content behind the URI, and the source of the redirect URI and other client data.

The AS SHOULD only automatically redirect the user agent if it trusts the redirect URI. If the URI is not trusted, the AS MAY inform the user and rely on the user to make the correct decision.

5. Acknowledgements

We would like to thank Brock Allen, Annabelle Richard Backman, Dominick Baier, Vittorio Bertocci, Brian Campbell, William Dennis, George Fletscher, Dick Hardt, Joseph Heenan, Pedram Hosseyni, Phil Hunt, Jared Jennings, Michael B. Jones, Konstantin Lapine, Neil Madden, Christian Mainka, Jim Manico, Nov Matake, Doug McDorman, Karsten Meyer zu Selhausen, Aaron Parecki, Michael Peck, Johan Peeters, Nat Sakimura, Guido Schmitz, Travis Spencer, Petteri Stenius, Tomek Stojecki, Tim Wuertele, David Waite and Hans Zandbelt for their valuable feedback.

6. IANA Considerations

This draft makes no requests to IANA.

7. Security Considerations

Security considerations are described in Section 2, Section 3, and Section 4.

8. Normative References

[RFC6819] Lodderstedt, T., Ed., McGloin, M., and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations", RFC 6819, DOI 10.17487/RFC6819, January 2013, .

[RFC8414] Jones, M., Sakimura, N., and J. Bradley, "OAuth 2.0 Authorization Server Metadata", RFC 8414, DOI 10.17487/RFC8414, June 2018, .

[RFC8252] Denniss, W. and J. Bradley, "OAuth 2.0 for Native Apps", BCP 212, RFC 8252, DOI 10.17487/RFC8252, October 2017, .

[RFC8705] Campbell, B., Bradley, J., Sakimura, N., and T. Lodderstedt, "OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens", February 2020, .

[RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, .

[OpenID.Discovery] Sakimura, N., Bradley, J., Jones, M., and E. Jay, "OpenID Connect Discovery 1.0 incorporating errata set 1", 8 November 2014, .

[RFC9068] Bertocci, V., "JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens", RFC 9068, DOI 10.17487/RFC9068, October 2021, .

[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, .

[OpenID.Core]
Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0 incorporating errata set 1", 8 November 2014,
.

[RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012,
.

[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,
.

9. Informative References

[I-D.bradley-oauth-jwt-encoded-state]
Bradley, J., Lodderstedt, D. T., and H. Zandbelt, "Encoding claims in the OAuth 2 state parameter using a JWT", Work in Progress, Internet-Draft, draft-bradley-oauth-jwt-encoded-state-09, 4 November 2018,
.

[RFC8473] Popov, A., Nystroem, M., Balfanz, D., Ed., Harper, N., and J. Hodges, "Token Binding over HTTP", RFC 8473, DOI 10.17487/RFC8473, October 2018,
.

[arXiv.1901.11520]
Fett, D., Hosseyni, P., and R. Küsters, "An Extensive Formal Security Analysis of the OpenID Financial-grade API", 31 January 2019, .

[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,
.

[CSP-2] West, M., Barth, A., and D. Veditz, "Content Security Policy Level 2", July 2015, .

[JARM] Lodderstedt, T. and B. Campbell, "Financial-grade API: JWT Secured Authorization Response Mode for OAuth 2.0 (JARM)", 17 October 2018,
.

[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, .

[I-D.ietf-oauth-dpop] Fett, D., Campbell, B., Bradley, J., Lodderstedt, T., Jones, M., and D. Waite, "OAuth 2.0 Demonstrating Proof-of-Possession at the Application Layer (DPoP)", Work in Progress, Internet-Draft, draft-ietf-oauth-dpop-11, 10 August 2022, .

[WebCrypto] Watson, M., "Web Cryptography API", 26 January 2017, .

[subdomaintakeover] Liu, D., Hao, S., and H. Wang, "All Your DNS Records Point to Us: Understanding the Security Threats of Dangling DNS Records", 24 October 2016, .

[RFC9126] Lodderstedt, T., Campbell, B., Sakimura, N., Tonge, D., and F. Skokan, "OAuth 2.0 Pushed Authorization Requests", RFC 9126, DOI 10.17487/RFC9126, September 2021, .

[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, .

[I-D.ietf-oauth-token-binding] Jones, M. B., Campbell, B., Bradley, J., and W. Denniss, "OAuth 2.0 Token Binding", Work in Progress, Internet-Draft, draft-ietf-oauth-token-binding-08, 19 October 2018, .

[oauth_security_ubc] Sun, S.-T. and K. Beznosov, "The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems", October 2012, .

[oauth_security_jcs_14]
Bansal, C., Bhargavan, K., Delignat-Lavaud, A., and S. Maffeis, "Discovering concrete attacks on website authorization by formal analysis", 23 April 2014, .

[owasp_redir]
"OWASP Cheat Sheet Series - Unvalidated Redirects and Forwards", .

[WebAuthn] Balfanz, D., Czeskis, A., Hodges, J., Jones, J.C., Jones, M.B., Kumar, A., Liao, A., Lindemann, R., and E. Lundberg, "Web Authentication: An API for accessing Public Key Credentials Level 1", 4 March 2019, .

[arXiv.1601.01229]
Fett, D., Küsters, R., and G. Schmitz, "A Comprehensive Formal Security Analysis of OAuth 2.0", 6 January 2016, .

[RFC9101] Sakimura, N., Bradley, J., and M. Jones, "The OAuth 2.0 Authorization Framework: JWT-Secured Authorization Request (JAR)", RFC 9101, DOI 10.17487/RFC9101, August 2021, .

[webappsec-referrer-policy]
Eisinger, J. and E. Stark, "Referrer Policy", 20 April 2017, .

[oauth_security_cmu]
Chen, E., Pei, Y., Chen, S., Tian, Y., Kotcher, R., and P. Tague, "OAuth Demystified for Mobile Application Developers", November 2014, .

[I-D.ietf-oauth-iss-auth-resp]
Selhausen, K. M. Z. and D. Fett, "OAuth 2.0 Authorization Server Issuer Identification", Work in Progress, Internet-Draft, draft-ietf-oauth-iss-auth-resp-05, 11 January 2022, .

[bug.chromium]
"Referer header includes URL fragment when opening link using New Tab",
.

[OAuth.Post]
Jones, M. and B. Campbell, "OAuth 2.0 Form Post Response Mode", 27 April 2015, .

[I-D.ietf-oauth-pop-key-distribution]
Bradley, J., Hunt, P., Jones, M. B., Tschofenig, H., and M. Meszaros, "OAuth 2.0 Proof-of-Possession: Authorization Server to Client Key Distribution", Work in Progress, Internet-Draft, draft-ietf-oauth-pop-key-distribution-07, 27 March 2019, .

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997,
.

[I-D.ietf-oauth-rar]
Lodderstedt, T., Richer, J., and B. Campbell, "OAuth 2.0 Rich Authorization Requests", Work in Progress, Internet-Draft, draft-ietf-oauth-rar-12, 5 May 2022,
.

[I-D.ietf-oauth-signed-http-request]
Richer, J., Bradley, J., and H. Tschofenig, "A Method for Signing HTTP Requests for OAuth", Work in Progress, Internet-Draft, draft-ietf-oauth-signed-http-request-03, 8 August 2016, .

[I-D.sakimura-oauth-jpop]
Sakimura, N., Li, K., and J. Bradley, "The OAuth 2.0 Authorization Framework: JWT Pop Token Usage", Work in Progress, Internet-Draft, draft-sakimura-oauth-jpop-05, 22 July 2019, .

[RFC9207] Meyer zu Selhausen, K. and D. Fett, "OAuth 2.0 Authorization Server Issuer Identification", RFC 9207, DOI 10.17487/RFC9207, March 2022, .

[RFC8707] Campbell, B., Bradley, J., and H. Tschofenig, "Resource Indicators for OAuth 2.0", RFC 8707, DOI 10.17487/RFC8707, February 2020, .

[arXiv.1704.08539] Fett, D., Küsters, R., and G. Schmitz, "The Web SSO Standard OpenID Connect: In-Depth Formal Security Analysis and Security Guidelines", 27 April 2017, .

[arXiv.1508.04324v2] Mladenov, V., Mainka, C., and J. Schwenk, "On the security of modern Single Sign-On Protocols: Second-Order Vulnerabilities in OpenID Connect", 7 January 2016, .

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, .

Appendix A. Document History

[[To be removed from the final specification]]

-20

* Improved description of authorization code injection attacks and PKCE protection

* Removed recommendation for MTLS in discussion (not reflected in actual Recommendations section)

* Reworded "placeholder" text in security considerations.

* Alphabetized list of names and fixed unicode problem

* Explained Clickjacking

* Explained Open Redirectors

* Clarified references to attacker model by including a link to Section 3

* Clarified description of "CSRF tokens" and reference to RFC6819

* Described that OIDC can prevent access token injection

* Updated references

-19

* Changed affiliation of Andrey Labunets

* Editorial change to clarify the new recommendations for refresh tokens

-18

* Fix editorial and spelling issues.

* Change wording for disallowing HTTP redirect URIs.

-17

* Make the use of metadata RECOMMENDED for both servers and clients

* Make announcing PKCE support in metadata the RECOMMENDED way (before: either metadata or deployment-specific way)

* AS also MUST NOT expose open redirectors.

* Mention that attackers can collaborate.

* Update recommendations regarding mix-up defense, building upon [I-D.ietf-oauth-iss-auth-resp].

* Improve description of mix-up attack.

* Make HTTPS mandatory for most redirect URIs.

-16

* Make MTLS a suggestion, not RECOMMENDED.

* Add important requirements when using nonce for code injection protection.

* Highlight requirements for refresh token sender-constraining.

* Make PKCE a MUST for public clients.

* Describe PKCE Downgrade Attacks and countermeasures.

* Allow variable port numbers in localhost redirect URIs as in RFC8252, Section 7.3.

-15

* Update reference to DPoP

* Fix reference to RFC8414

* Move to xml2rfcv3

-14

* Added info about using CSP to prevent clickjacking

* Changes from WGLC feedback

* Editorial changes

* AS MUST announce PKCE support either in metadata or using deployment-specific ways (before: SHOULD)

-13

* Discourage use of Resource Owner Password Credentials Grant

* Added text on client impersonating resource owner

* Recommend asymmetric methods for client authentication

* Encourage use of PKCE mode "S256"

* PKCE may replace state for CSRF protection

* AS SHOULD publish PKCE support

* Cleaned up discussion on auth code injection

* AS MUST support PKCE

-12

* Added updated attacker model

-11

* Adapted section 2.1.2 to outcome of consensus call

* more text on refresh token inactivity and implementation note on refresh token replay detection via refresh token rotation

-10

* incorporated feedback by Joseph Heenan

* changed occurrences of SHALL to MUST

* added text on lack of token/cert binding support tokens issued in the authorization response as justification to not recommend issuing tokens there at all

* added requirement to authenticate clients during code exchange (PKCE or client credential) to 2.1.1.

* added section on refresh tokens

* editorial enhancements to 2.1.2 based on feedback

-09

* changed text to recommend not to use implicit but code

* added section on access token injection

* reworked sections 3.1 through 3.3 to be more specific on implicit grant issues

-08

* added recommendations re implicit and token injection

* uppercased key words in Section 2 according to RFC 2119

-07

* incorporated findings of Doug McDorman

* added section on HTTP status codes for redirects

* added new section on access token privilege restriction based on comments from Johan Peeters

-06

* reworked section 3.8.1

* incorporated Phil Hunt's feedback

* reworked section on mix-up

* extended section on code leakage via referrer header to also cover state leakage

* added Daniel Fett as author

* replaced text intended to inform WG discussion by recommendations to implementors

* modified example URLs to conform to RFC 2606

-05

* Completed sections on code leakage via referrer header, attacks in browser, mix-up, and CSRF

* Reworked Code Injection Section

* Added reference to OpenID Connect spec

* removed refresh token leakage as respective considerations have been given in section 10.4 of RFC 6749

* first version on open redirection

* incorporated Christian Mainka's review feedback

-04

* Restructured document for better readability

* Added best practices on Token Leakage prevention

-03

* Added section on Access Token Leakage at Resource Server

* incorporated Brian Campbell's findings

-02

* Folded Mix up and Access Token leakage through a bad AS into new section for dynamic OAuth threats

* reworked dynamic OAuth section

-01

* Added references to mitigation methods for token leakage

* Added reference to Token Binding for Authorization Code

* incorporated feedback of Phil Hunt

* fixed numbering issue in attack descriptions in section 2

-00 (WG document)

* turned the ID into a WG document and a BCP

* Added federated app login as topic in Other Topics

Authors' Addresses

Torsten Lodderstedt
yes.com
Email: torsten@lodderstedt.net

John Bradley
Yubico
Email: ve7jtb@ve7jtb.com

Andrey Labunets
Independent Researcher
Email: isciurus@gmail.com

Daniel Fett
yes.com
Email: mail@danielfett.de

Chapter 8

RFC 8628: OAuth 2.0 Device Authorization Grant

The Device Flow is an extension that enables devices with no browser or limited input capability to obtain access tokens. You'll typically see this on devices like an Apple TV where there is no web browser, or on "internet of things" devices where there is no input mechanism other than a few buttons.

The flow works by having users visit a URL on a secondary device like a smartphone and entering a code that is shown on the device. The device can accomplish the flow without the need for a browser or interacting with the user in any way other than getting them to visit the URL and enter a code.

This flow is most widely seen on smart TVs, although there are many more creative applications of it as well, such as using it in cars or for command line applications.

Internet Engineering Task Force (IETF)
Request for Comments: 8628
Category: Standards Track
ISSN: 2070-1721

W. Denniss
Google
J. Bradley
Ping Identity
M. Jones
Microsoft
H. Tschofenig
ARM Limited

August 2019

OAuth 2.0 Device Authorization Grant

Abstract

The OAuth 2.0 device authorization grant is designed for Internet-connected devices that either lack a browser to perform a user-agent-based authorization or are input constrained to the extent that requiring the user to input text in order to authenticate during the authorization flow is impractical. It enables OAuth clients on such devices (like smart TVs, media consoles, digital picture frames, and printers) to obtain user authorization to access protected resources by using a user agent on a separate device.

Status of This Memo

This is an Internet Standards Track document.

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

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc8628.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction 3
2. Terminology 5
3. Protocol 5
 3.1. Device Authorization Request 5
 3.2. Device Authorization Response 7
 3.3. User Interaction 8
  3.3.1. Non-Textual Verification URI Optimization 9
 3.4. Device Access Token Request 10
 3.5. Device Access Token Response 11
4. Discovery Metadata 12
5. Security Considerations 12
 5.1. User Code Brute Forcing 12
 5.2. Device Code Brute Forcing 13
 5.3. Device Trustworthiness 13
 5.4. Remote Phishing 14
 5.5. Session Spying 15
 5.6. Non-Confidential Clients 15
 5.7. Non-Visual Code Transmission 15
6. Usability Considerations 16
 6.1. User Code Recommendations 16
 6.2. Non-Browser User Interaction 17
7. IANA Considerations 17
 7.1. OAuth Parameter Registration 17
 7.2. OAuth URI Registration 17
 7.3. OAuth Extensions Error Registration 18
 7.4. OAuth Authorization Server Metadata 18
8. Normative References 19
Acknowledgements 20
Authors' Addresses 21

1. Introduction

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

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

The operating requirements for using this authorization grant type are:

(1) The device is already connected to the Internet.

(2) The device is able to make outbound HTTPS requests.

(3) The device is able to display or otherwise communicate a URI and code sequence to the user.

(4) The user has a secondary device (e.g., personal computer or smartphone) from which they can process the request.

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

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

The device client typically chooses the set of authorization servers to support (i.e., its own authorization server or those of providers with which it has relationships). It is common for the device client to support only one authorization server, such as in the case of a TV application for a specific media provider that supports only that media provider's authorization server. The user may not yet have an established relationship with that authorization provider, though one can potentially be set up during the authorization flow.

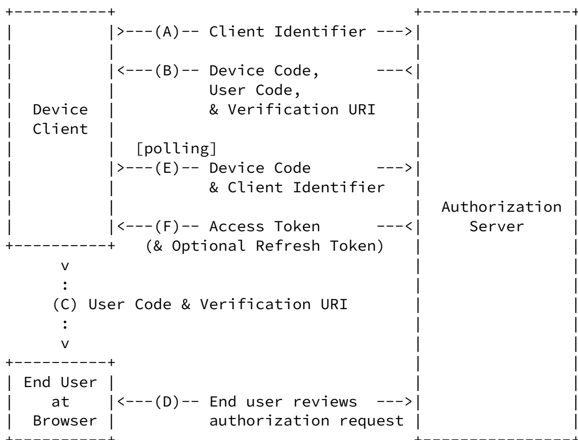


Figure 1: Device Authorization Flow

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

(A) The client requests access from the authorization server and includes its client identifier in the request.

(B) The authorization server issues a device code and an end-user code and provides the end-user verification URI.

(C) The client instructs the end user to use a user agent on another device and visit the provided end-user verification URI. The client provides the user with the end-user code to enter in order to review the authorization request.

(D) The authorization server authenticates the end user (via the user agent), and prompts the user to input the user code provided by the device client. The authorization server validates the user code provided by the user, and prompts the user to accept or decline the request.

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

(F) The authorization server validates the device code provided by the client and responds with the access token if the client is granted access, an error if they are denied access, or an indication that the client should continue to poll.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Protocol

3.1. Device Authorization Request

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

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

The client makes a device authorization request to the device authorization endpoint by including the following parameters using the "application/x-www-form-urlencoded" format, per Appendix B of [RFC6749], with a character encoding of UTF-8 in the HTTP request entity-body:

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

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

For example, the client makes the following HTTPS request:

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

client_id=1406020730>ope=example_scope
```

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

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

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

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

3.2. Device Authorization Response

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

device_code
: REQUIRED. The device verification code.

user_code
: REQUIRED. The end-user verification code.

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

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

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

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

For example:

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

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

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

3.3. User Interaction

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

```
+-----------------------------------------------+
|                                               |
|  Using a browser on another device, visit:    |
|  https://example.com/device                   |
|                                               |
|  And enter the code:                          |
|  WDJB-MJHT                                    |
|                                               |
+-----------------------------------------------+
```

Figure 2: Example User Instruction

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

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

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

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

3.3.1. Non-Textual Verification URI Optimization

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

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

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

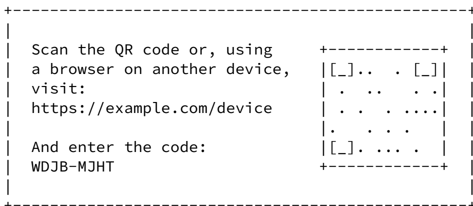


Figure 3: Example User Instruction with QR Code Representation of the Complete Verification URI

3.4. Device Access Token Request

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

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

device_code
: REQUIRED. The device verification code, "device_code" from the device authorization response, defined in Section 3.2.

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

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

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

grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Adevice_code
&device_code=GmRhmhcxhwAzkoEqiMEg_DnyEysNkuNhszIySk9eS
&client_id=1406020730
```

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

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

3.5. Device Access Token Response

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

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

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

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

access_denied
: The authorization request was denied.

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

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

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

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

4. Discovery Metadata

Support for this protocol is declared in OAuth 2.0 Authorization Server Metadata [RFC8414] as follows. The value "urn:ietf:params:oauth:grant-type:device_code" is included in values of the "grant_types_supported" key, and the following new key value pair is added:

device_authorization_endpoint
: OPTIONAL. URL of the authorization server's device authorization endpoint, as defined in Section 3.1.

5. Security Considerations

5.1. User Code Brute Forcing

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

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

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

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

5.2. Device Code Brute Forcing

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

5.3. Device Trustworthiness

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

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

5.4. Remote Phishing

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

For authorization servers that support the "verification_uri_complete" optimization discussed in Section 3.3.1, it is particularly important to confirm that the device is in the user's possession, as the user no longer has to type in the code being displayed on the device manually. One suggestion is to display the code during the authorization flow and ask the user to verify that the same code is currently 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, log in, etc.) but should be sufficiently short to limit the usability of a code obtained for phishing. This doesn't prevent a phisher from presenting a fresh token, particularly if they are interacting with the user in real time, but it does limit the viability of codes sent over email or text message.

5.5. Session Spying

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

5.6. Non-Confidential Clients

Device clients are generally incapable of maintaining the confidentiality of their credentials, as users in possession of the device can reverse-engineer it and extract the credentials. Therefore, unless additional measures are taken, they should be treated as public clients (as defined by Section 2.1 of [RFC6749]), which are susceptible to impersonation. The security considerations of Section 5.3.1 of [RFC6819] and Sections 8.5 and 8.6 of [RFC8252] apply to such clients.

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

5.7. Non-Visual Code Transmission

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

6. Usability Considerations

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

6.1. User Code Recommendations

For many users, their nearest Internet-connected device will be their mobile phone; 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), the limitations of such devices should be taken into account when selecting the user code character set.

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

An example user code following this guideline, "WDJB-MJHT", contains 8 significant characters and has dashes added for end-user readability. The resulting entropy is 20^8.

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

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

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

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

6.2. Non-Browser User Interaction

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

7. IANA Considerations

7.1. OAuth Parameter Registration

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

Name: device_code
Parameter Usage Location: token request
Change Controller: IESG
Reference: Section 3.4 of RFC 8628

7.2. OAuth URI Registration

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

URN: urn:ietf:params:oauth:grant-type:device_code
Common Name: Device Authorization Grant Type for OAuth 2.0
Change Controller: IESG
Specification Document: Section 3.4 of RFC 8628

7.3. OAuth Extensions Error Registration

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

Name: authorization_pending
Usage Location: Token endpoint response
Protocol Extension: RFC 8628
Change Controller: IETF
Reference: Section 3.5 of RFC 8628

Name: access_denied
Usage Location: Token endpoint response
Protocol Extension: RFC 8628
Change Controller: IETF
Reference: Section 3.5 of RFC 8628

Name: slow_down
Usage Location: Token endpoint response
Protocol Extension: RFC 8628
Change Controller: IETF
Reference: Section 3.5 of RFC 8628

Name: expired_token
Usage Location: Token endpoint response
Protocol Extension: RFC 8628
Change Controller: IETF
Reference: Section 3.5 of RFC 8628

7.4. OAuth Authorization Server Metadata

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

Metadata name: device_authorization_endpoint
Metadata Description: URL of the authorization server's device authorization endpoint
Change Controller: IESG
Reference: Section 4 of RFC 8628

8. Normative References

[IANA.OAuth.Parameters]
IANA, "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,
.

[RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012,
.

[RFC6755] Campbell, B. and H. Tschofenig, "An IETF URN Sub-Namespace for OAuth", RFC 6755, DOI 10.17487/RFC6755, October 2012,
.

[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,
.

[RFC7525] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", BCP 195, RFC 7525, DOI 10.17487/RFC7525, May 2015, .

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, .

[RFC8252] Denniss, W. and J. Bradley, "OAuth 2.0 for Native Apps", BCP 212, RFC 8252, DOI 10.17487/RFC8252, October 2017,
.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017,
.

[RFC8414] Jones, M., Sakimura, N., and J. Bradley, "OAuth 2.0 Authorization Server Metadata", RFC 8414, DOI 10.17487/RFC8414, June 2018,
.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, .

Acknowledgements

The starting point for this document was the Internet-Draft draft-recordon-oauth-v2-device, authored by David Recordon and Brent Goldman, which itself was based on content in draft versions of the OAuth 2.0 protocol specification removed prior to publication due to a then-lack of sufficient deployment expertise. Thank you to the OAuth Working Group members who contributed to those earlier drafts.

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

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

Ben Campbell, Brian Campbell, Roshni Chandrashekhar, Alissa Cooper, Eric Fazendin, Benjamin Kaduk, Jamshid Khosravian, Mirja Kuehlewind, Torsten Lodderstedt, James Manger, Dan McNulty, Breno de Medeiros, Alexey Melnikov, Simon Moffatt, Stein Myrseth, Emond Papegaaij, Justin Richer, Adam Roach, Nat Sakimura, Andrew Sciberras, Marius Scurtescu, Filip Skokan, Robert Sparks, Ken Wang, Christopher Wood, Steven E. Wright, and Qin Wu.

Authors' Addresses

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

Email: wdenniss@google.com
URI: https://wdenniss.com/deviceflow

John Bradley
Ping Identity

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

Michael B. Jones
Microsoft

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

Hannes Tschofenig
ARM Limited
Austria

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

Chapter 9

RFC 7009: OAuth 2.0 Token Revocation

Token Revocation describes a new endpoint on the authorization server that clients can use to notify the server that an access token or refresh token is no longer needed. This is used to enable a “log out” feature in clients, allowing the authorization server to clean up any tokens or other data associated with that session.

Internet Engineering Task Force (IETF)
Request for Comments: 7009
Category: Standards Track
ISSN: 2070-1721

T. Lodderstedt, Ed.
Deutsche Telekom AG
S. Dronia

M. Scurtescu
Google

August 2013

OAuth 2.0 Token Revocation

Abstract

This document proposes an additional endpoint for OAuth authorization servers, which allows clients to notify the authorization server that a previously obtained refresh or access token is no longer needed. This allows the authorization server to clean up security credentials. A revocation request will invalidate the actual token and, if applicable, other tokens based on the same authorization grant.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc7009.

Copyright Notice

Copyright (c) 2013 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 Language 3
2. Token Revocation 3
2.1. Revocation Request 4
2.2. Revocation Response 5
2.2.1. Error Response 6
2.3. Cross-Origin Support 6
3. Implementation Note 7
4. IANA Considerations 8
4.1. OAuth Extensions Error Registration 8
4.1.1. The "unsupported_token_type" Error Value 8
4.1.2. OAuth Token Type Hints Registry 8
4.1.2.1. Registration Template 9
4.1.2.2. Initial Registry Contents 9
5. Security Considerations 9
6. Acknowledgements 10
7. References 10
7.1. Normative References 10
7.2. Informative References 11

1. Introduction

The OAuth 2.0 core specification [RFC6749] defines several ways for a client to obtain refresh and access tokens. This specification supplements the core specification with a mechanism to revoke both types of tokens. A token is a string representing an authorization grant issued by the resource owner to the client. A revocation request will invalidate the actual token and, if applicable, other tokens based on the same authorization grant and the authorization grant itself.

From an end-user's perspective, OAuth is often used to log into a certain site or application. This revocation mechanism allows a client to invalidate its tokens if the end-user logs out, changes identity, or uninstalls the respective application. Notifying the authorization server that the token is no longer needed allows the authorization server to clean up data associated with that token (e.g., session data) and the underlying authorization grant. This behavior prevents a situation in which there is still a valid authorization grant for a particular client of which the end-user is not aware. This way, token revocation prevents abuse of abandoned tokens and facilitates a better end-user experience since invalidated authorization grants will no longer turn up in a list of authorization grants the authorization server might present to the end-user.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2. Token Revocation

Implementations MUST support the revocation of refresh tokens and SHOULD support the revocation of access tokens (see Implementation Note).

The client requests the revocation of a particular token by making an HTTP POST request to the token revocation endpoint URL. This URL MUST conform to the rules given in [RFC6749], Section 3.1. Clients MUST verify that the URL is an HTTPS URL.

The means to obtain the location of the revocation endpoint is out of the scope of this specification. For example, the client developer may consult the server's documentation or automatic discovery may be used. As this endpoint is handling security credentials, the endpoint location needs to be obtained from a trustworthy source.

Since requests to the token revocation endpoint result in the transmission of plaintext credentials in the HTTP request, URLs for token revocation endpoints MUST be HTTPS URLs. The authorization server MUST use Transport Layer Security (TLS) [RFC5246] in a version compliant with [RFC6749], Section 1.6. Implementations MAY also support additional transport-layer security mechanisms that meet their security requirements.

If the host of the token revocation endpoint can also be reached over HTTP, then the server SHOULD also offer a revocation service at the corresponding HTTP URI, but it MUST NOT publish this URI as a token revocation endpoint. This ensures that tokens accidentally sent over HTTP will be revoked.

2.1. Revocation Request

The client constructs the request by including the following parameters using the "application/x-www-form-urlencoded" format in the HTTP request entity-body:

token REQUIRED. The token that the client wants to get revoked.

token_type_hint OPTIONAL. A hint about the type of the token submitted for revocation. Clients MAY pass this parameter in order to help the authorization server to optimize the token lookup. If the server is unable to locate the token using the given hint, it MUST extend its search across all of its supported token types. An authorization server MAY ignore this parameter, particularly if it is able to detect the token type automatically. This specification defines two such values:

* access_token: An access token as defined in [RFC6749], Section 1.4

* refresh_token: A refresh token as defined in [RFC6749], Section 1.5

Specific implementations, profiles, and extensions of this specification MAY define other values for this parameter using the registry defined in Section 4.1.2.

The client also includes its authentication credentials as described in Section 2.3. of [RFC6749].

For example, a client may request the revocation of a refresh token with the following request:

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

token=45ghiukldjahdnhzdauz&token_type_hint=refresh_token
```

The authorization server first validates the client credentials (in case of a confidential client) and then verifies whether the token was issued to the client making the revocation request. If this validation fails, the request is refused and the client is informed of the error by the authorization server as described below.

In the next step, the authorization server invalidates the token. The invalidation takes place immediately, and the token cannot be used again after the revocation. In practice, there could be a propagation delay, for example, in which some servers know about the invalidation while others do not. Implementations should minimize that window, and clients must not try to use the token after receipt of an HTTP 200 response from the server.

Depending on the authorization server's revocation policy, the revocation of a particular token may cause the revocation of related tokens and the underlying authorization grant. If the particular token is a refresh token and the authorization server supports the revocation of access tokens, then the authorization server SHOULD also invalidate all access tokens based on the same authorization grant (see Implementation Note). If the token passed to the request is an access token, the server MAY revoke the respective refresh token as well.

Note: A client compliant with [RFC6749] must be prepared to handle unexpected token invalidation at any time. Independent of the revocation mechanism specified in this document, resource owners may revoke authorization grants, or the authorization server may invalidate tokens in order to mitigate security threats. Thus, having different server policies with respect to cascading the revocation of tokens should not pose interoperability problems.

2.2. Revocation Response

The authorization server responds with HTTP status code 200 if the token has been revoked successfully or if the client submitted an invalid token.

Note: invalid tokens do not cause an error response since the client cannot handle such an error in a reasonable way. Moreover, the purpose of the revocation request, invalidating the particular token, is already achieved.

The content of the response body is ignored by the client as all necessary information is conveyed in the response code.

An invalid token type hint value is ignored by the authorization server and does not influence the revocation response.

2.2.1. Error Response

The error presentation conforms to the definition in Section 5.2 of [RFC6749]. The following additional error code is defined for the token revocation endpoint:

unsupported_token_type: The authorization server does not support the revocation of the presented token type. That is, the client tried to revoke an access token on a server not supporting this feature.

If the server responds with HTTP status code 503, the client must assume the token still exists and may retry after a reasonable delay. The server may include a "Retry-After" header in the response to indicate how long the service is expected to be unavailable to the requesting client.

2.3. Cross-Origin Support

The revocation endpoint MAY support Cross-Origin Resource Sharing (CORS) [W3C.WD-cors-20120403] if it is aimed at use in combination with user-agent-based applications.

In addition, for interoperability with legacy user-agents, it MAY also offer JSONP (Remote JSON - JSONP) [jsonp] by allowing GET requests with an additional parameter:

callback OPTIONAL. The qualified name of a JavaScript function.

For example, a client may request the revocation of an access token with the following request (line breaks are for display purposes only):

```
https://example.com/revoke?token=agabcdefddddafdd&
callback=package.myCallback
```

Successful response:

```
package.myCallback();
```

Error response:

```
package.myCallback({"error":"unsupported_token_type"});
```

Clients should be aware that when relying on JSONP, a malicious revocation endpoint may attempt to inject malicious code into the client.

3. Implementation Note

OAuth 2.0 allows deployment flexibility with respect to the style of access tokens. The access tokens may be self-contained so that a resource server needs no further interaction with an authorization server issuing these tokens to perform an authorization decision of the client requesting access to a protected resource. A system design may, however, instead use access tokens that are handles referring to authorization data stored at the authorization server. This consequently requires a resource server to issue a request to the respective authorization server to retrieve the content of the access token every time a client presents an access token.

While these are not the only options, they illustrate the implications for revocation. In the latter case, the authorization server is able to revoke an access token previously issued to a client when the resource server relays a received access token. In the former case, some (currently non-standardized) backend interaction between the authorization server and the resource server may be used when immediate access token revocation is desired. Another design alternative is to issue short-lived access tokens, which can be refreshed at any time using the corresponding refresh tokens. This allows the authorization server to impose a limit on the time revoked when access tokens are in use.

Which approach of token revocation is chosen will depend on the overall system design and on the application service provider's risk analysis. The cost of revocation in terms of required state and communication overhead is ultimately the result of the desired security properties.

4. IANA Considerations

This specification registers an error value in the "OAuth Extensions Error Registry" and establishes the "OAuth Token Type Hints" registry.

4.1. OAuth Extensions Error Registration

This specification registers the following error value in the "OAuth Extensions Error Registry" defined in [RFC6749].

4.1.1. The "unsupported_token_type" Error Value

Error name: unsupported_token_type

Error Usage Location: Revocation endpoint error response

Related Protocol Extension: Token Revocation Endpoint

Change controller: IETF

Specification document(s): [RFC7009]

4.1.2. OAuth Token Type Hints Registry

This specification establishes the "OAuth Token Type Hints" registry. Possible values of the parameter "token_type_hint" (see Section 2.1) are registered with a Specification Required ([RFC5226]) after a two-week review period on the oauth-ext-review@ietf.org mailing list, on the advice of one or more Designated Experts. However, to allow for the allocation of values prior to publication, the Designated Expert(s) may approve registration once they are satisfied that such a specification will be published. Registration requests must be sent to the oauth-ext-review@ietf.org mailing list for review and comment, with an appropriate subject (e.g., "Request for parameter: example"). Within the review period, the Designated Expert(s) will either approve or deny the registration request, communicating this decision to the review list and IANA. Denials should include an explanation and, if applicable, suggestions as to how to make the request successful. IANA must only accept registry updates from the Designated Expert(s) and should direct all requests for registration to the review mailing list.

4.1.2.1. Registration Template

Hint Value: The additional value, which can be used to indicate a certain token type to the authorization server.

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

Specification document(s): Reference to the document(s) that specifies the type, preferably including a URI that can be used to retrieve a copy of the document(s). An indication of the relevant sections may also be included but is not required.

4.1.2.2. Initial Registry Contents

The OAuth Token Type Hint registry's initial contents are as follows.

Hint Value	Change Controller	Reference
access_token	IETF	[RFC7009]
refresh_token	IETF	[RFC7009]

Table 1: OAuth Token Type Hints initial registry contents

5. Security Considerations

If the authorization server does not support access token revocation, access tokens will not be immediately invalidated when the corresponding refresh token is revoked. Deployments must take this into account when conducting their security risk analysis.

Cleaning up tokens using revocation contributes to overall security and privacy since it reduces the likelihood for abuse of abandoned tokens. This specification in general does not intend to provide countermeasures against token theft and abuse. For a discussion of respective threats and countermeasures, consult the security considerations given in Section 10 of the OAuth core specification [RFC6749] and the OAuth threat model document [RFC6819].

Malicious clients could attempt to use the new endpoint to launch denial-of-service attacks on the authorization server. Appropriate countermeasures, which should be in place for the token endpoint as well, MUST be applied to the revocation endpoint (see [RFC6819], Section 4.4.1.11). Specifically, invalid token type hints may

misguide the authorization server and cause additional database lookups. Care MUST be taken to prevent malicious clients from exploiting this feature to launch denial-of-service attacks.

A malicious client may attempt to guess valid tokens on this endpoint by making revocation requests against potential token strings. According to this specification, a client's request must contain a valid client_id, in the case of a public client, or valid client credentials, in the case of a confidential client. The token being revoked must also belong to the requesting client. If an attacker is able to successfully guess a public client's client_id and one of their tokens, or a private client's credentials and one of their tokens, they could do much worse damage by using the token elsewhere than by revoking it. If they chose to revoke the token, the legitimate client will lose its authorization grant and will need to prompt the user again. No further damage is done and the guessed token is now worthless.

Since the revocation endpoint is handling security credentials, clients need to obtain its location from a trustworthy source only. Otherwise, an attacker could capture valid security tokens by utilizing a counterfeit revocation endpoint. Moreover, in order to detect counterfeit revocation endpoints, clients MUST authenticate the revocation endpoint (certificate validation, etc.).

6. Acknowledgements

We would like to thank Peter Mauritius, Amanda Anganes, Mark Wubben, Hannes Tschofenig, Michiel de Jong, Doug Foiles, Paul Madsen, George Fletcher, Sebastian Ebling, Christian Stuebner, Brian Campbell, Igor Faynberg, Lukas Rosenstock, and Justin Richer for their valuable feedback.

7. References

7.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

[RFC6749] Hardt, D., "The OAuth 2.0 Authorization Framework", RFC 6749, October 2012.

7.2. Informative References

[RFC6819] Lodderstedt, T., McGloin, M., and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations", RFC 6819, January 2013.

[W3C.WD-cors-20120403] Kesteren, A., "Cross-Origin Resource Sharing", World Wide Web Consortium LastCall WD-cors-20120403, April 2012, .

[jsonp] Ippolito, B., "Remote JSON - JSONP", December 2005, .

Authors' Addresses

Torsten Lodderstedt (editor)
Deutsche Telekom AG

EMail: torsten@lodderstedt.net

Stefanie Dronia

EMail: sdronia@gmx.de

Marius Scurtescu
Google

EMail: mscurtescu@google.com

Chapter 10

RFC 7662: OAuth 2.0 Token Introspection

The Token Introspection spec defines a mechanism for resource servers to obtain information about access tokens. Without this spec, resource servers have to have a bespoke way of checking whether access tokens are valid, and finding out information needed in order to process the request. This typically would occur by having the resource server and authorization server share a database, or by agreeing on an access token format such as JWT and sharing encryption or signing keys.

With this spec, resource servers can check the validity of access tokens and find out other information with an HTTP API call, leading to better separation of concerns between the authorization server and any resource servers.

Internet Engineering Task Force (IETF)
Request for Comments: 7662
Category: Standards Track
ISSN: 2070-1721

J. Richer, Ed.
October 2015

OAuth 2.0 Token Introspection

Abstract

This specification defines a method for a protected resource to query an OAuth 2.0 authorization server to determine the active state of an OAuth 2.0 token and to determine meta-information about this token. OAuth 2.0 deployments can use this method to convey information about the authorization context of the token from the authorization server to the protected resource.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc7662.

Copyright Notice

Copyright (c) 2015 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. Notational Conventions 3
1.2. Terminology 3
2. Introspection Endpoint 3
2.1. Introspection Request 4
2.2. Introspection Response 6
2.3. Error Response 8
3. IANA Considerations 9
3.1. OAuth Token Introspection Response Registry 9
3.1.1. Registration Template 10
3.1.2. Initial Registry Contents 10
4. Security Considerations 12
5. Privacy Considerations 14
6. References 15
6.1. Normative References 15
6.2. Informative References 16
Appendix A. Use with Proof-of-Possession Tokens 17
Acknowledgements 17
Author's Address 17

1. Introduction

In OAuth 2.0 [RFC6749], the contents of tokens are opaque to clients. This means that the client does not need to know anything about the content or structure of the token itself, if there is any. However, there is still a large amount of metadata that may be attached to a token, such as its current validity, approved scopes, and information about the context in which the token was issued. These pieces of information are often vital to protected resources making authorization decisions based on the tokens being presented. Since OAuth 2.0 does not define a protocol for the resource server to learn meta-information about a token that it has received from an authorization server, several different approaches have been developed to bridge this gap. These include using structured token formats such as JWT [RFC7519] or proprietary inter-service communication mechanisms (such as shared databases and protected enterprise service buses) that convey token information.

This specification defines a protocol that allows authorized protected resources to query the authorization server to determine the set of metadata for a given token that was presented to them by an OAuth 2.0 client. This metadata includes whether or not the token is currently active (or if it has expired or otherwise been revoked), what rights of access the token carries (usually conveyed through OAuth 2.0 scopes), and the authorization context in which the token was granted (including who authorized the token and which client it

was issued to). Token introspection allows a protected resource to query this information regardless of whether or not it is carried in the token itself, allowing this method to be used along with or independently of structured token values. Additionally, a protected resource can use the mechanism described in this specification to introspect the token in a particular authorization decision context and ascertain the relevant metadata about the token to make this authorization decision appropriately.

1.1. 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 [RFC2119].

Unless otherwise noted, all the protocol parameter names and values are case sensitive.

1.2. Terminology

This section defines the terminology used by this specification. This section is a normative portion of this specification, imposing requirements upon implementations.

This specification uses the terms "access token", "authorization endpoint", "authorization grant", "authorization server", "client", "client identifier", "protected resource", "refresh token", "resource owner", "resource server", and "token endpoint" defined by OAuth 2.0 [RFC6749], and the terms "claim names" and "claim values" defined by JSON Web Token (JWT) [RFC7519].

This specification defines the following terms:

Token Introspection
: The act of inquiring about the current state of an OAuth 2.0 token through use of the network protocol defined in this document.

Introspection Endpoint
: The OAuth 2.0 endpoint through which the token introspection operation is accomplished.

2. Introspection Endpoint

The introspection endpoint is an OAuth 2.0 endpoint that takes a parameter representing an OAuth 2.0 token and returns a JSON [RFC7159] document representing the meta information surrounding the token, including whether this token is currently active. The

definition of an active token is dependent upon the authorization server, but this is commonly a token that has been issued by this authorization server, is not expired, has not been revoked, and is valid for use at the protected resource making the introspection call.

The introspection endpoint MUST be protected by a transport-layer security mechanism as described in Section 4. The means by which the protected resource discovers the location of the introspection endpoint are outside the scope of this specification.

2.1. Introspection Request

The protected resource calls the introspection endpoint using an HTTP POST [RFC7231] request with parameters sent as "application/x-www-form-urlencoded" data as defined in [W3C.REC-html5-20141028]. The protected resource sends a parameter representing the token along with optional parameters representing additional context that is known by the protected resource to aid the authorization server in its response.

token
: REQUIRED. The string value of the token. For access tokens, this is the "access_token" value returned from the token endpoint defined in OAuth 2.0 [RFC6749], Section 5.1. For refresh tokens, this is the "refresh_token" value returned from the token endpoint as defined in OAuth 2.0 [RFC6749], Section 5.1. Other token types are outside the scope of this specification.

token_type_hint
: OPTIONAL. A hint about the type of the token submitted for introspection. The protected resource MAY pass this parameter to help the authorization server optimize the token lookup. If the server is unable to locate the token using the given hint, it MUST extend its search across all of its supported token types. An authorization server MAY ignore this parameter, particularly if it is able to detect the token type automatically. Values for this field are defined in the "OAuth Token Type Hints" registry defined in OAuth Token Revocation [RFC7009].

The introspection endpoint MAY accept other OPTIONAL parameters to provide further context to the query. For instance, an authorization server may desire to know the IP address of the client accessing the protected resource to determine if the correct client is likely to be presenting the token. The definition of this or any other parameters are outside the scope of this specification, to be defined by service documentation or extensions to this specification. If the authorization server is unable to determine the state of the token

without additional information, it SHOULD return an introspection response indicating the token is not active as described in Section 2.2.

To prevent token scanning attacks, the endpoint MUST also require some form of authorization to access this endpoint, such as client authentication as described in OAuth 2.0 [RFC6749] or a separate OAuth 2.0 access token such as the bearer token described in OAuth 2.0 Bearer Token Usage [RFC6750]. The methods of managing and validating these authentication credentials are out of scope of this specification.

For example, the following shows a protected resource calling the token introspection endpoint to query about an OAuth 2.0 bearer token. The protected resource is using a separate OAuth 2.0 bearer token to authorize this call.

The following is a non-normative example request:

```
POST /introspect HTTP/1.1
Host: server.example.com
Accept: application/json
Content-Type: application/x-www-form-urlencoded
Authorization: Bearer 23410913-abewfq.123483

token=2YotnFZFEjr1zCsicMWpAA
```

In this example, the protected resource uses a client identifier and client secret to authenticate itself to the introspection endpoint. The protected resource also sends a token type hint indicating that it is inquiring about an access token.

The following is a non-normative example request:

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

token=mF_9.B5f-4.1JqM&token_type_hint=access_token
```

2.2. Introspection Response

The server responds with a JSON object [RFC7159] in "application/json" format with the following top-level members.

active
: REQUIRED. Boolean indicator of whether or not the presented token is currently active. The specifics of a token's "active" state will vary depending on the implementation of the authorization server and the information it keeps about its tokens, but a "true" value return for the "active" property will generally indicate that a given token has been issued by this authorization server, has not been revoked by the resource owner, and is within its given time window of validity (e.g., after its issuance time and before its expiration time). See Section 4 for information on implementation of such checks.

scope
: OPTIONAL. A JSON string containing a space-separated list of scopes associated with this token, in the format described in Section 3.3 of OAuth 2.0 [RFC6749].

client_id
: OPTIONAL. Client identifier for the OAuth 2.0 client that requested this token.

username
: OPTIONAL. Human-readable identifier for the resource owner who authorized this token.

token_type
: OPTIONAL. Type of the token as defined in Section 5.1 of OAuth 2.0 [RFC6749].

exp
: OPTIONAL. Integer timestamp, measured in the number of seconds since January 1 1970 UTC, indicating when this token will expire, as defined in JWT [RFC7519].

iat
: OPTIONAL. Integer timestamp, measured in the number of seconds since January 1 1970 UTC, indicating when this token was originally issued, as defined in JWT [RFC7519].

nbf
: OPTIONAL. Integer timestamp, measured in the number of seconds since January 1 1970 UTC, indicating when this token is not to be used before, as defined in JWT [RFC7519].

sub
OPTIONAL. Subject of the token, as defined in JWT [RFC7519]. Usually a machine-readable identifier of the resource owner who authorized this token.

aud
OPTIONAL. Service-specific string identifier or list of string identifiers representing the intended audience for this token, as defined in JWT [RFC7519].

iss
OPTIONAL. String representing the issuer of this token, as defined in JWT [RFC7519].

jti
OPTIONAL. String identifier for the token, as defined in JWT [RFC7519].

Specific implementations MAY extend this structure with their own service-specific response names as top-level members of this JSON object. Response names intended to be used across domains MUST be registered in the "OAuth Token Introspection Response" registry defined in Section 3.1.

The authorization server MAY respond differently to different protected resources making the same request. For instance, an authorization server MAY limit which scopes from a given token are returned for each protected resource to prevent a protected resource from learning more about the larger network than is necessary for its operation.

The response MAY be cached by the protected resource to improve performance and reduce load on the introspection endpoint, but at the cost of liveness of the information used by the protected resource to make authorization decisions. See Section 4 for more information regarding the trade off when the response is cached.

For example, the following response contains a set of information about an active token:

The following is a non-normative example response:

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

{
 "active": true,
 "client_id": "l238j323ds-23ij4",
 "username": "jdoe",
 "scope": "read write dolphin",
 "sub": "Z5O3upPC88QrAjx00dis",
 "aud": "https://protected.example.net/resource",
 "iss": "https://server.example.com/",
 "exp": 1419356238,
 "iat": 1419350238,
 "extension_field": "twenty-seven"
}
```

If the introspection call is properly authorized but the token is not active, does not exist on this server, or the protected resource is not allowed to introspect this particular token, then the authorization server MUST return an introspection response with the "active" field set to "false". Note that to avoid disclosing too much of the authorization server's state to a third party, the authorization server SHOULD NOT include any additional information about an inactive token, including why the token is inactive.

The following is a non-normative example response for a token that has been revoked or is otherwise invalid:

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

{
 "active": false
}
```

2.3. Error Response

If the protected resource uses OAuth 2.0 client credentials to authenticate to the introspection endpoint and its credentials are invalid, the authorization server responds with an HTTP 401 (Unauthorized) as described in Section 5.2 of OAuth 2.0 [RFC6749].

If the protected resource uses an OAuth 2.0 bearer token to authorize its call to the introspection endpoint and the token used for authorization does not contain sufficient privileges or is otherwise invalid for this request, the authorization server responds with an HTTP 401 code as described in Section 3 of OAuth 2.0 Bearer Token Usage [RFC6750].

Note that a properly formed and authorized query for an inactive or otherwise invalid token (or a token the protected resource is not allowed to know about) is not considered an error response by this specification. In these cases, the authorization server MUST instead respond with an introspection response with the "active" field set to "false" as described in Section 2.2.

3. IANA Considerations

3.1. OAuth Token Introspection Response Registry

This specification establishes the "OAuth Token Introspection Response" registry.

OAuth registration client metadata names and descriptions are registered by Specification Required [RFC5226] after a two-week review period on the oauth-ext-review@ietf.org mailing list, on the advice of one or more Designated Experts. However, to allow for the allocation of names prior to publication, the Designated Expert(s) may approve registration once they are satisfied that such a specification will be published.

Registration requests sent to the mailing list for review should use an appropriate subject (e.g., "Request to register OAuth Token Introspection Response name: example").

Within the review period, the Designated Expert(s) will either approve or deny the registration request, communicating this decision to the review list and IANA. Denials should include an explanation and, if applicable, suggestions as to how to make the request successful.

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

3.1.1. Registration Template

Name:
The name requested (e.g., "example"). This name is case sensitive. Names that match other registered names in a case insensitive manner SHOULD NOT be accepted. Names that match claims registered in the "JSON Web Token Claims" registry established by [RFC7519] SHOULD have comparable definitions and semantics.

Description:
Brief description of the metadata value (e.g., "Example description").

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

Specification document(s):
Reference to the document(s) that specify the token endpoint authorization method, preferably including a URI that can be used to retrieve a copy of the document(s). An indication of the relevant sections may also be included but is not required.

3.1.2. Initial Registry Contents

The initial contents of the "OAuth Token Introspection Response" registry are as follows:

- Name: "active"
- Description: Token active status
- Change Controller: IESG
- Specification Document(s): Section 2.2 of RFC 7662 (this document).

- Name: "username"
- Description: User identifier of the resource owner
- Change Controller: IESG
- Specification Document(s): Section 2.2 of RFC 7662 (this document).

- Name: "client_id"
- Description: Client identifier of the client
- Change Controller: IESG
- Specification Document(s): Section 2.2 of RFC 7662 (this document).

- o Name: "scope"
- o Description: Authorized scopes of the token
- o Change Controller: IESG
- o Specification Document(s): Section 2.2 of RFC 7662 (this document).

- o Name: "token_type"
- o Description: Type of the token
- o Change Controller: IESG
- o Specification Document(s): Section 2.2 of RFC 7662 (this document).

- o Name: "exp"
- o Description: Expiration timestamp of the token
- o Change Controller: IESG
- o Specification Document(s): Section 2.2 of RFC 7662 (this document).

- o Name: "iat"
- o Description: Issuance timestamp of the token
- o Change Controller: IESG
- o Specification Document(s): Section 2.2 of RFC 7662 (this document).

- o Name: "nbf"
- o Description: Timestamp before which the token is not valid
- o Change Controller: IESG
- o Specification Document(s): Section 2.2 of RFC 7662 (this document).

- o Name: "sub"
- o Description: Subject of the token
- o Change Controller: IESG
- o Specification Document(s): Section 2.2 of RFC 7662 (this document).

- o Name: "aud"
- o Description: Audience of the token
- o Change Controller: IESG
- o Specification Document(s): Section 2.2 of RFC 7662 (this document).

- o Name: "iss"
- o Description: Issuer of the token
- o Change Controller: IESG
- o Specification Document(s): Section 2.2 of RFC 7662 (this document).

- o Name: "jti"
- o Description: Unique identifier of the token
- o Change Controller: IESG
- o Specification Document(s): Section 2.2 of RFC 7662 (this document).

4. Security Considerations

Since there are many different and valid ways to implement an OAuth 2.0 system, there are consequently many ways for an authorization server to determine whether or not a token is currently "active". However, since resource servers using token introspection rely on the authorization server to determine the state of a token, the authorization server MUST perform all applicable checks against a token's state. For instance, these tests include the following:

- o If the token can expire, the authorization server MUST determine whether or not the token has expired.
- o If the token can be issued before it is able to be used, the authorization server MUST determine whether or not a token's valid period has started yet.
- o If the token can be revoked after it was issued, the authorization server MUST determine whether or not such a revocation has taken place.
- o If the token has been signed, the authorization server MUST validate the signature.
- o If the token can be used only at certain resource servers, the authorization server MUST determine whether or not the token can be used at the resource server making the introspection call.

If an authorization server fails to perform any applicable check, the resource server could make an erroneous security decision based on that response. Note that not all of these checks will be applicable to all OAuth 2.0 deployments and it is up to the authorization server to determine which of these checks (and any other checks) apply.

If left unprotected and un-throttled, the introspection endpoint could present a means for an attacker to poll a series of possible token values, fishing for a valid token. To prevent this, the authorization server MUST require authentication of protected resources that need to access the introspection endpoint and SHOULD require protected resources to be specifically authorized to call the introspection endpoint. The specifics of such authentication credentials are out of scope of this specification, but commonly these credentials could take the form of any valid client authentication mechanism used with the token endpoint, an OAuth 2.0 access token, or other HTTP authorization or authentication mechanism. A single piece of software acting as both a client and a

protected resource MAY reuse the same credentials between the token endpoint and the introspection endpoint, though doing so potentially conflates the activities of the client and protected resource portions of the software and the authorization server MAY require separate credentials for each mode.

Since the introspection endpoint takes in OAuth 2.0 tokens as parameters and responds with information used to make authorization decisions, the server MUST support Transport Layer Security (TLS) 1.2 [RFC5246] and MAY support additional transport-layer mechanisms meeting its security requirements. When using TLS, the client or protected resource MUST perform a TLS/SSL server certificate check, as specified in [RFC6125]. Implementation security considerations can be found in Recommendations for Secure Use of TLS and DTLS [BCP195].

To prevent the values of access tokens from leaking into server-side logs via query parameters, an authorization server offering token introspection MAY disallow the use of HTTP GET on the introspection endpoint and instead require the HTTP POST method to be used at the introspection endpoint.

To avoid disclosing the internal state of the authorization server, an introspection response for an inactive token SHOULD NOT contain any additional claims beyond the required "active" claim (with its value set to "false").

Since a protected resource MAY cache the response of the introspection endpoint, designers of an OAuth 2.0 system using this protocol MUST consider the performance and security trade-offs inherent in caching security information such as this. A less aggressive cache with a short timeout will provide the protected resource with more up-to-date information (due to it needing to query the introspection endpoint more often) at the cost of increased network traffic and load on the introspection endpoint. A more aggressive cache with a longer duration will minimize network traffic and load on the introspection endpoint, but at the risk of stale information about the token. For example, the token may be revoked while the protected resource is relying on the value of the cached response to make authorization decisions. This creates a window during which a revoked token could be used at the protected resource. Consequently, an acceptable cache validity duration needs to be carefully considered given the concerns and sensitivities of the protected resource being accessed and the likelihood of a token being revoked or invalidated in the interim period. Highly sensitive environments can opt to disable caching entirely on the protected resource to eliminate the risk of stale cached information entirely, again at the cost of increased network traffic and server load. If

the response contains the "exp" parameter (expiration), the response MUST NOT be cached beyond the time indicated therein.

An authorization server offering token introspection must be able to understand the token values being presented to it during this call. The exact means by which this happens is an implementation detail and is outside the scope of this specification. For unstructured tokens, this could take the form of a simple server-side database query against a data store containing the context information for the token. For structured tokens, this could take the form of the server parsing the token, validating its signature or other protection mechanisms, and returning the information contained in the token back to the protected resource (allowing the protected resource to be unaware of the token's contents, much like the client). Note that for tokens carrying encrypted information that is needed during the introspection process, the authorization server must be able to decrypt and validate the token to access this information. Also note that in cases where the authorization server stores no information about the token and has no means of accessing information about the token by parsing the token itself, it cannot likely offer an introspection service.

5. Privacy Considerations

The introspection response may contain privacy-sensitive information such as user identifiers for resource owners. When this is the case, measures MUST be taken to prevent disclosure of this information to unintended parties. One method is to transmit user identifiers as opaque service-specific strings, potentially returning different identifiers to each protected resource.

If the protected resource sends additional information about the client's request to the authorization server (such as the client's IP address) using an extension of this specification, such information could have additional privacy considerations that the extension should detail. However, the nature and implications of such extensions are outside the scope of this specification.

Omitting privacy-sensitive information from an introspection response is the simplest way of minimizing privacy issues.

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, .

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, DOI 10.17487/RFC5226, May 2008, .

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, .

[RFC6125] Saint-Andre, P. and J. Hodges, "Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS)", RFC 6125, DOI 10.17487/RFC6125, March 2011, .

[RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, .

[RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, .

[RFC7009] Lodderstedt, T., Ed., Dronia, S., and M. Scurtescu, "OAuth 2.0 Token Revocation", RFC 7009, DOI 10.17487/RFC7009, August 2013, .

[RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March 2014, .

[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, .

[RFC7519] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, .

[W3C.REC-html5-20141028] Hickson, I., Berjon, R., Faulkner, S., Leithead, T., Navara, E., O'Connor, E., and S. Pfeiffer, "HTML5", World Wide Web Consortium Recommendation REC-html5-20141028, October 2014, .

6.2. Informative References

[BCP195] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", BCP 195, RFC 7525, May 2015, .

Appendix A. Use with Proof-of-Possession Tokens

With bearer tokens such as those defined by OAuth 2.0 Bearer Token Usage [RFC6750], the protected resource will have in its possession the entire secret portion of the token for submission to the introspection service. However, for proof-of-possession style tokens, the protected resource will have only a token identifier used during the request, along with the cryptographic signature on the request. To validate the signature on the request, the protected resource could be able to submit the token identifier to the authorization server's introspection endpoint to obtain the necessary key information needed for that token. The details of this usage are outside the scope of this specification and will be defined in an extension to this specification in concert with the definition of proof-of-possession tokens.

Acknowledgements

Thanks to the OAuth Working Group and the User Managed Access Working Group for feedback and review of this document, and to the various implementors of both the client and server components of this specification. In particular, the author would like to thank Amanda Anganes, John Bradley, Thomas Broyer, Brian Campbell, George Fletcher, Paul Freemantle, Thomas Hardjono, Eve Maler, Josh Mandel, Steve Moore, Mike Schwartz, Prabath Siriwardena, Sarah Squire, and Hannes Tschofennig.

Author's Address

Justin Richer (editor)

Email: ietf@justin.richer.org

Chapter 11

RFC 8414: OAuth 2.0 Authorization Server Metadata

The Authorization Server Metadata spec (also known as OAuth Discovery) defines an endpoint clients can use to look up the information needed to interact with a particular OAuth server. This includes things like finding the authorization and token endpoints, listing the supported scopes and response types, and providing access token signing keys to resource servers.

Internet Engineering Task Force (IETF)
Request for Comments: 8414
Category: Standards Track
ISSN: 2070-1721

M. Jones
Microsoft
N. Sakimura
NRI
J. Bradley
Yubico

June 2018

OAuth 2.0 Authorization Server Metadata

Abstract

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

Status of This Memo

This is an Internet Standards Track document.

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

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc8414.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction 2
   1.1. Requirements Notation and Conventions 3
   1.2. Terminology 3
2. Authorization Server Metadata 4
   2.1. Signed Authorization Server Metadata 8
3. Obtaining Authorization Server Metadata 8
   3.1. Authorization Server Metadata Request 9
   3.2. Authorization Server Metadata Response 10
   3.3. Authorization Server Metadata Validation 11
4. String Operations 11
5. Compatibility Notes 11
6. Security Considerations 12
   6.1. TLS Requirements 12
   6.2. Impersonation Attacks 12
   6.3. Publishing Metadata in a Standard Format 13
   6.4. Protected Resources 13
7. IANA Considerations 14
   7.1. OAuth Authorization Server Metadata Registry 14
        7.1.1. Registration Template 15
        7.1.2. Initial Registry Contents 16
   7.2. Updated Registration Instructions 19
   7.3. Well-Known URI Registry 19
        7.3.1. Registry Contents 19
8. References 20
   8.1. Normative References 20
   8.2. Informative References 22
Acknowledgements 23
Authors' Addresses 23

1. Introduction

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

The metadata for an authorization server is retrieved from a well-known location as a JSON [RFC8259] document, which declares its endpoint locations and authorization server capabilities. This process is described in Section 3.

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

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

1.1. Requirements Notation and Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

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

1.2. Terminology

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

2. Authorization Server Metadata

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

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

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

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

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

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

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

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

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

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

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

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

service_documentation
OPTIONAL. URL of a page containing human-readable information that developers might want or need to know when using the authorization server. In particular, if the authorization server

does not support Dynamic Client Registration, then information on how to register clients needs to be provided in this documentation.

ui_locales_supported
OPTIONAL. Languages and scripts supported for the user interface, represented as a JSON array of language tag values from BCP 47 [RFC5646]. If omitted, the set of supported languages and scripts is unspecified.

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

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

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

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

revocation_endpoint_auth_signing_alg_values_supported
OPTIONAL. JSON array containing a list of the JWS signing algorithms ("alg" values) supported by the revocation endpoint for the signature on the JWT [JWT] used to authenticate the client at

the revocation endpoint for the "private_key_jwt" and "client_secret_jwt" authentication methods. This metadata entry MUST be present if either of these authentication methods are specified in the "revocation_endpoint_auth_methods_supported" entry. No default algorithms are implied if this entry is omitted. The value "none" MUST NOT be used.

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

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

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

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

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

2.1. Signed Authorization Server Metadata

In addition to JSON elements, metadata values MAY also be provided as a "signed_metadata" value, which is a JSON Web Token (JWT) [JWT] that asserts metadata values about the authorization server as a bundle. A set of claims that can be used in signed metadata is defined in Section 2. The signed metadata MUST be digitally signed or MACed using JSON Web Signature (JWS) [JWS] and MUST contain an "iss" (issuer) claim denoting the party attesting to the claims in the signed metadata. Consumers of the metadata MAY ignore the signed metadata if they do not support this feature. If the consumer of the metadata supports signed metadata, metadata values conveyed in the signed metadata MUST take precedence over the corresponding values conveyed using plain JSON elements.

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

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

3. Obtaining Authorization Server Metadata

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

Different applications utilizing OAuth authorization servers in application-specific ways may define and register different well-known URI suffixes used to publish authorization server metadata as used by those applications. For instance, if the example application uses an OAuth authorization server in an example-specific way, and there are example-specific metadata values that it needs to publish, then it might register and use the "example-configuration" URI suffix and publish the metadata document at the path formed by inserting "/.well-known/example-configuration" between the host and path components of the authorization server's issuer identifier. Alternatively, many such applications will use the default well-known

URI string "/.well-known/oauth-authorization-server", which is the right choice for general-purpose OAuth authorization servers, and not register an application-specific one.

An OAuth 2.0 application using this specification MUST specify what well-known URI suffix it will use for this purpose. The same authorization server MAY choose to publish its metadata at multiple well-known locations derived from its issuer identifier, for example, publishing metadata at both "/.well-known/example-configuration" and "/.well-known/oauth-authorization-server".

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

3.1. Authorization Server Metadata Request

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

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

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

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

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

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

3.2. Authorization Server Metadata Response

The response is a set of claims about the authorization server's configuration, including all necessary endpoints and public key location information. A successful response MUST use the 200 OK HTTP status code and return a JSON object using the "application/json" content type that contains a set of claims as its members that are a subset of the metadata values defined in Section 2. Other claims MAY also be returned.

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

An error response uses the applicable HTTP status code value.

The following is a non-normative example response:

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

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

3.3. Authorization Server Metadata Validation

The "issuer" value returned MUST be identical to the authorization server's issuer identifier value into which the well-known URI string was inserted to create the URL used to retrieve the metadata. If these values are not identical, the data contained in the response MUST NOT be used.

4. String Operations

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

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

1. Remove any JSON-applied escaping to produce an array of Unicode code points.

2. Unicode Normalization [USA15] MUST NOT be applied at any point to either the JSON string or the string it is to be compared against.

3. Comparisons between the two strings MUST be performed as a Unicode code-point-to-code-point equality comparison.

Note that this is the same equality comparison procedure described in Section 8.3 of [RFC8259].

5. Compatibility Notes

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

The algorithm for transforming the issuer identifier to an authorization server metadata location defined in Section 3 is equivalent to the corresponding transformation defined in Section 4 of "OpenID Connect Discovery 1.0" [OpenID.Discovery], provided that the issuer identifier contains no path component. However, they are different when there is a path component, because OpenID Connect

Discovery 1.0 specifies that the well-known URI string is appended to the issuer identifier (e.g., "https://example.com/issuer1/.well-known/openid-configuration"), whereas this specification specifies that the well-known URI string is inserted before the path component of the issuer identifier (e.g., "https://example.com/.well-known/openid-configuration/issuer1").

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

6. Security Considerations

6.1. TLS Requirements

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

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

6.2. Impersonation Attacks

TLS certificate checking MUST be performed by the client, as described in Section 6.1, when making an authorization server metadata request. Checking that the server certificate is valid for the issuer identifier URL prevents man-in-middle and DNS-based

attacks. These attacks could cause a client to be tricked into using an attacker's keys and endpoints, which would enable impersonation of the legitimate authorization server. If an attacker can accomplish this, they can access the resources that the affected client has access to using the authorization server that they are impersonating.

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

6.3. Publishing Metadata in a Standard Format

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

6.4. Protected Resources

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

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

7. IANA Considerations

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

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

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

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

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

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

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

7.1. OAuth Authorization Server Metadata Registry

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

The Designated Experts must either:

(a) require that metadata names and values being registered use only printable ASCII characters excluding double quote ('"') and backslash ('\') (the Unicode characters with code points U+0021, U+0023 through U+005B, and U+005D through U+007E), or

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

7.1.1. Registration Template

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

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

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

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

7.1.2. Initial Registry Contents

- Metadata Name: issuer
- Metadata Description: Authorization server's issuer identifier URL
- Change Controller: IESG
- Specification Document(s): Section 2 of RFC 8414

- Metadata Name: authorization_endpoint
- Metadata Description: URL of the authorization server's authorization endpoint
- Change Controller: IESG
- Specification Document(s): Section 2 of RFC 8414

- Metadata Name: token_endpoint
- Metadata Description: URL of the authorization server's token endpoint
- Change Controller: IESG
- Specification Document(s): Section 2 of RFC 8414

- Metadata Name: jwks_uri
- Metadata Description: URL of the authorization server's JWK Set document
- Change Controller: IESG
- Specification Document(s): Section 2 of RFC 8414

- Metadata Name: registration_endpoint
- Metadata Description: URL of the authorization server's OAuth 2.0 Dynamic Client Registration Endpoint
- Change Controller: IESG
- Specification Document(s): Section 2 of RFC 8414

- Metadata Name: scopes_supported
- Metadata Description: JSON array containing a list of the OAuth 2.0 "scope" values that this authorization server supports
- Change Controller: IESG
- Specification Document(s): Section 2 of RFC 8414

- Metadata Name: response_types_supported
- Metadata Description: JSON array containing a list of the OAuth 2.0 "response_type" values that this authorization server supports
- Change Controller: IESG
- Specification Document(s): Section 2 of RFC 8414

- Metadata Name: response_modes_supported
- Metadata Description: JSON array containing a list of the OAuth 2.0 "response_mode" values that this authorization server supports
- Change Controller: IESG
- Specification Document(s): Section 2 of RFC 8414

- o Metadata Name: grant_types_supported
- o Metadata Description: JSON array containing a list of the OAuth 2.0 grant type values that this authorization server supports
- o Change Controller: IESG
- o Specification Document(s): Section 2 of RFC 8414

- o Metadata Name: token_endpoint_auth_methods_supported
- o Metadata Description: JSON array containing a list of client authentication methods supported by this token endpoint
- o Change Controller: IESG
- o Specification Document(s): Section 2 of RFC 8414

- o Metadata Name: token_endpoint_auth_signing_alg_values_supported
- o Metadata Description: JSON array containing a list of the JWS signing algorithms supported by the token endpoint for the signature on the JWT used to authenticate the client at the token endpoint
- o Change Controller: IESG
- o Specification Document(s): Section 2 of RFC 8414

- o Metadata Name: service_documentation
- o Metadata Description: URL of a page containing human-readable information that developers might want or need to know when using the authorization server
- o Change Controller: IESG
- o Specification Document(s): Section 2 of RFC 8414

- o Metadata Name: ui_locales_supported
- o Metadata Description: Languages and scripts supported for the user interface, represented as a JSON array of language tag values from BCP 47
- o Change Controller: IESG
- o Specification Document(s): Section 2 of RFC 8414

- o Metadata Name: op_policy_uri
- o Metadata Description: URL that the authorization server provides to the person registering the client to read about the authorization server's requirements on how the client can use the data provided by the authorization server
- o Change Controller: IESG
- o Specification Document(s): Section 2 of RFC 8414

- o Metadata Name: op_tos_uri
- o Metadata Description: URL that the authorization server provides to the person registering the client to read about the authorization server's terms of service
- o Change Controller: IESG
- o Specification Document(s): Section 2 of RFC 8414

- o Metadata Name: revocation_endpoint
- o Metadata Description: URL of the authorization server's OAuth 2.0 revocation endpoint
- o Change Controller: IESG
- o Specification Document(s): Section 2 of RFC 8414

- o Metadata Name: revocation_endpoint_auth_methods_supported
- o Metadata Description: JSON array containing a list of client authentication methods supported by this revocation endpoint
- o Change Controller: IESG
- o Specification Document(s): Section 2 of RFC 8414

- o Metadata Name: revocation_endpoint_auth_signing_alg_values_supported
- o Metadata Description: JSON array containing a list of the JWS signing algorithms supported by the revocation endpoint for the signature on the JWT used to authenticate the client at the revocation endpoint
- o Change Controller: IESG
- o Specification Document(s): Section 2 of RFC 8414

- o Metadata Name: introspection_endpoint
- o Metadata Description: URL of the authorization server's OAuth 2.0 introspection endpoint
- o Change Controller: IESG
- o Specification Document(s): Section 2 of RFC 8414

- o Metadata Name: introspection_endpoint_auth_methods_supported
- o Metadata Description: JSON array containing a list of client authentication methods supported by this introspection endpoint
- o Change Controller: IESG
- o Specification Document(s): Section 2 of RFC 8414

- o Metadata Name: introspection_endpoint_auth_signing_alg_values_supported
- o Metadata Description: JSON array containing a list of the JWS signing algorithms supported by the introspection endpoint for the signature on the JWT used to authenticate the client at the introspection endpoint
- o Change Controller: IESG
- o Specification Document(s): Section 2 of RFC 8414

- o Metadata Name: code_challenge_methods_supported
- o Metadata Description: PKCE code challenge methods supported by this authorization server
- o Change Controller: IESG
- o Specification Document(s): Section 2 of RFC 8414

- Metadata Name: signed_metadata
- Metadata Description: Signed JWT containing metadata values about the authorization server as claims
- Change Controller: IESG
- Specification Document(s): Section 2.1 of RFC 8414

7.2. Updated Registration Instructions

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

- OAuth Access Token Types
- OAuth Token Endpoint Authentication Methods

IANA has added a link to this specification in the Reference sections of these registries.

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

7.3. Well-Known URI Registry

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

7.3.1. Registry Contents

- URI suffix: oauth-authorization-server
- Change controller: IESG
- Specification document: Section 3 of RFC 8414
- Related information: (none)

8. References

8.1. Normative References

[BCP195] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", BCP 195, RFC 7525, May 2015, .

[IANA.OAuth.Parameters] IANA, "OAuth Parameters", .

[JWE] Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)", RFC 7516, DOI 10.17487/RFC7516, May 2015, .

[JWK] Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7517, May 2015, .

[JWS] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, .

[JWT] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, .

[OAuth.Post] Jones, M. and B. Campbell, "OAuth 2.0 Form Post Response Mode", April 2015, .

[OAuth.Responses] de Medeiros, B., Ed., Scurtescu, M., Tarjan, P., and M. Jones, "OAuth 2.0 Multiple Response Type Encoding Practices", February 2014, .

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, .

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, .

[RFC5646] Phillips, A., Ed. and M. Davis, Ed., "Tags for Identifying Languages", BCP 47, RFC 5646, DOI 10.17487/RFC5646, September 2009, .

[RFC5785] Nottingham, M. and E. Hammer-Lahav, "Defining Well-Known Uniform Resource Identifiers (URIs)", RFC 5785, DOI 10.17487/RFC5785, April 2010, .

[RFC6125] Saint-Andre, P. and J. Hodges, "Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS)", RFC 6125, DOI 10.17487/RFC6125, March 2011, .

[RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, .

[RFC7009] Lodderstedt, T., Ed., Dronia, S., and M. Scurtescu, "OAuth 2.0 Token Revocation", RFC 7009, DOI 10.17487/RFC7009, August 2013, .

[RFC7033] Jones, P., Salgueiro, G., Jones, M., and J. Smarr, "WebFinger", RFC 7033, DOI 10.17487/RFC7033, September 2013, .

[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, .

[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, .

[RFC7662] Richer, J., Ed., "OAuth 2.0 Token Introspection", RFC 7662, DOI 10.17487/RFC7662, October 2015, .

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, .

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, .

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, .

[UNICODE] The Unicode Consortium, "The Unicode Standard", .

[USA15] Davis, M., Ed. and K. Whistler, Ed., "Unicode Normalization Forms", Unicode Standard Annex #15, May 2018, .

8.2. Informative References

[IANA.well-known] IANA, "Well-Known URIs", .

[MIX-UP] Jones, M., Bradley, J., and N. Sakimura, "OAuth 2.0 Mix-Up Mitigation", Work in Progress, draft-ietf-oauth-mix-up-mitigation-01, July 2016.

[OpenID.Core] Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0", November 2014, .

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

[OpenID.Registration] Sakimura, N., Bradley, J., and M. Jones, "OpenID Connect Dynamic Client Registration 1.0", November 2014, .

Acknowledgements

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

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

Authors' Addresses

Michael B. Jones
Microsoft

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

Nat Sakimura
Nomura Research Institute, Ltd.

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

John Bradley
Yubico

Email: RFC8414@ve7jtb.com
URI: http://www.thread-safe.com/

Advanced Extensions

If you've made it this far, congrats! OAuth is a big space, and there's a lot more to cover than what we fit into this book! So far we've covered the most common and widely deployed specs in the space. But OAuth is never finished! There's a lot of exciting new developments happening in the working group still.

RFC 9126: Pushed Authorization Requests

https://datatracker.ietf.org/doc/html/rfc9126

Pushed Authorization Requests is a significant change to the OAuth flow to rely less on the front channel, by moving the start of the authorization code flow to the back channel instead.

RFC 9101: JWT Authorization Request

https://datatracker.ietf.org/doc/html/rfc9101

JWT Authorization Request describes a way to encode and sign the authorization request parameters as a JWT instead of using plain query string components. This lets the authorization server be sure that a real OAuth application initiated a particular authorziation request and the request has not been forged or tampered with.

RFC 9068: JWT Profile for OAuth Access Tokens

https://datatracker.ietf.org/doc/html/rfc9068

The JWT Profile for OAuth Access Tokens defines a JWT-based format and vocabulary for access tokens based on the collective experience learned from several large deployments.

RFC 8705: Mutual TLS

https://datatracker.ietf.org/doc/html/rfc8705

Mutual TLS describes a way to use TLS certificates for client authentication as well as issuing certificate-bound access tokens. This is one way implementers are improving upon the idea of bearer tokens.

Draft: DPoP

https://datatracker.ietf.org/doc/html/draft-ietf-oauth-dpop

DPoP describes an alternative to Mutual TLS for issuing access tokens that are bound to a particular client. This version accomplishes that in the application layer rather than transport layer.

Draft: Rich Authorization Requests

https://datatracker.ietf.org/doc/html/draft-ietf-oauth-rar

Rich Authorization Requests describes way for apps to request permissions more fine-grained than the current OAuth "scope" mechanism can provide. This could be used, for example, to authorize a particular bank transfer.

Related Communities

While the core OAuth specs are developed within the IETF OAuth Working Group, many other communities have built extensions on top of OAuth as well. Some of the new work within the OAuth group has started in these outside communities and was brought into the group after some time.

OpenID Foundation

https://openid.net

OpenID Connect builds upon the OAuth framework to provide user identity to clients. OpenID Connect adds a new type of token, an ID Token, which communicates user data to the application. It also defines some additional endpoints for things like session management.

Financial-Grade API Working Group (FAPI)

https://openid.net/wg/fapi/

The Financial-Grade API WG develops extensions that harden OAuth to meet the security requirements of the financial industry. Some of their current specs are being brought into the OAuth Working Group.

Kantara Initiative

https://kantarainitiative.org

The Kantara Initiative publishes the User-Managed Access (UMA) extension, describing an OAuth flow that enables a user to grant access to other users rather than only other applications.

IndieWeb

https://indieweb.org

The IndieWeb community publishes IndieAuth (*indieauth.net*), a decentralized identity protocol built on OAuth that uses URLs to identify users and applications. It enables people to use a domain under their control as their online identity while signing in to applications online.

Transactional Authorization

https://oauth.xyz

Transactional Authorization is a new proposal for a spec that could eventually replace OAuth 2.0. The project is in very early stages, but discussions have been happening at and around the OAuth group meetings by many of the same members of the OAuth group.

Acknowledgments

Thanks to everyone who has participated in the development of the many OAuth specs over the years!

Below are the names of everyone who is acknowledged in all the RFCs in this collection.

Aaron Parecki, Adam Roach, Aiden Bell, Alastair Mair, Alexey Melnikov, Alissa Cooper, Allen Tom, Amanda Anganes, Amos Jeffries, Andre DeMarre, Andrew Arnott, Andrew Sciberras, Andy Smith, Andy Zmolek, Annabelle Backman, Annabelle Richard Backman, Anthony Nadalin, Ashish Jain, Axel Nenker, Axel Nennker, Barry Leiba, Ben Campbell, Ben Laurie, Ben Wiley Sittler, Benjamin Kaduk, Bill Fisher, Bill de hOra, Bill de hÓra, Blaine Cook, Breno de Medeiros, Brent Goldman, Brian Campbell, Brian Carpenter, Brian Eaton, Brian Ellin, Brian Slesinsky, Brock Allen, Casey Lucas, Chasen Le Hara, Chris Messina, Christian Mainka, Christian Stuebner, Christopher Thomas, Christopher Wood, Chuck Mortimore, Craig Heath, Dan McNulty, Daniel Fett, Darren Bounds, David Recordon, David Waite, Derek Atkins, Dick Hardt, Dirk Balfanz, Dominick Baier, Donald Eastlake, Doug Foiles, Doug McDorman, Doug Tangren, Eduardo Gueiros, Elwyn Davies, Emond Papegaaij, Eran Hammer, Eran Sandler, Eric Fazendin, Eric Rescorla, Eric Sachs, Erik Wahlstrom, Evan Gilbert, Eve Maler, Filip Skokan, Francisco Corella, Franklin Tse, George Fletcher, George Fletscher, Guido Schmitz, Haibin Song, Hannes Tschofenig, Hans Zandbelt, Henry S. Thompson, Hui-Lan Lu,

Iain McGinniss, Ignacio Fiorentino, Igor Faynberg,
James Manger, Jamshid Khosravian, Janak Amarasena,
Jared Jennings, Jeremy Suriel, Jim Manico, Johan Peeters,
John Bradley, John Kemp, John Panzer, Jonathan Sergent,
Joseph Heenan, Josh Mandel, Julian Reschke, Justin Hart,
Justin Richer, Justin Smith, Karl McGuinness,
Karsten Meyer zu Selhausen, Kathleen Moriarty,
Kellan Elliott-McCrea, Ken Wang, Konstantin Lapine,
Kristoffer Gronowski, Larry Halff, Laurence Miao,
Leah Culver, Leo Tohill, Lewis Adam, Lisa Dusseault,
Luca Frosini, Lukas Rosenstock, Luke Shepard,
Madjid Nakhjiri, Marcos Caceres, Marius Scurtescu,
Mark Atwood, Mark Kent, Mark McGloin, Mark Nottingham,
Mark Wubben, Michael Adams, Michael B. Jones,
Michael Peck, Michiel de Jong, Mike Jones, Mike Schwartz,
Mirja Kuehlewind, Naitik Shah, Nat Sakimura, Neil Madden,
Nick Walker, Niv Steingarten, Nov Matake, Paul Freemantle,
Paul Madsen, Paul Tarjan, Paul Walker, Pedram Hosseyni,
Peifung E. Lam, Peter Mauritius, Peter Saint-Andre,
Petteri Stenius, Phil Hunt, Philippe De Ryck,
Prabath Siriwardena, Prateek Mishra, Qin Wu,
Raffi Krikorian, Rahul Ravikumar, Rasmus Lerdorf,
Richard M. Conlan, Rob Richards, Rob Sayre, Robert Sparks,
Roger Crew, Roshni Chandrashekhar, Ryo Ito, Sam Quigley,
Samuel Erdtman, Sarah Squire, Scott Cantor, Scott Tomilson,
Sean Turner, Sebastian Ebling, Sergey Beryozkin,
Shane Weeden, Shwetha Bhandari, Simon Moffatt,
Skylar Woodward, Stein Myrseth, Stephen Farrell,
Steve Moore, Steven E. Wright, Sudhi Umarji, Takamichi Saito,
Terry Jones, Thomas Broyer, Thomas Hardjono, Tim Bray,
Tim Freeman, Tim Wuertele, Todd Sieling, Tomek Stojecki,
Tony Nadalin, Torsten Lodderstedt, Travis Spencer,
Vittorio Bertocci, Vlad Skvortsov, Vladimir Dzhuvinov,
Wesley Eddy, William Dennis, William Denniss,
William J. Mills, Wolter Eldering, Yannick Majoros,
Yaron Y. Goland, Zachary Zeltsan.

The Little Book of OAuth 2.0 RFCs

RFC 6749, RFC 6750, RFC 6819, RFC 7009, RFC 7636, RFC 7662, RFC 8252, RFC 8414, RFC 8628, Security BCP, Browser App BCP

This reference guide will help you understand the context of each RFC that is part of OAuth.

The Little Book of OAuth 2.0 RFCs is a compilation of IETF RFCs that make up the OAuth spec. It contains a complete reproduction of each RFC listed above, along with a short introduction setting the context for why each RFC is important.

Aaron Parecki is a member of the OAuth Working Group at the IETF and has contributed to a number of the OAuth specifications. His books and trainings help developers architect and build secure systems using the latest standards.

ISBN 9798607503956

Aaron Parecki
aaronpk.com
oauth2simplified.com