Enterprise Mission Tailored OAuth 2.0 Profile

The views, opinions and/or findings contained in this report are those of The MITRE Corporation and should not be construed as an official government position, policy, or decision, unless designated by other documentation.

Approved for Public Release; Distribution Unlimited. Public Release Case Number 19-3213

©2020 The MITRE Corporation.
All rights reserved.

Bedford, MA

Beth Abramowitz
Kelley Burgin
Tommy Farinelli
Neil McNab
Michael Peck
Mark Russell
Roger Westman

February 2020

This page intentionally left blank.

Table of Contents

1 Introduction 1
1.1 Requirements Notation and Convention 1
1.2 Terminology 1
1.3 Conformance 2
1.4 Environment Overview 2
1.5 Use Cases 3
1.5.1 User Authorization Delegation to a Web Application 3
1.5.2 User Authorization Delegation to a Native Application 5
1.5.3 User Authorization Delegation to a Browser-Embedded Client 7
1.5.4 Token Exchange by Protected Resources 7
1.6 Global Requirements 8
2 Client Profiles 9
2.1 Client Types 9
2.1.1 Confidential Client 9
2.1.2 Public Client 9
2.2 Connection to the Authorization Server 9
2.2.1 Discovery 10
2.2.2. Requests to the Authorization Endpoint 10
2.2.3. Requests to the Token Endpoint 11
2.2.4 Client Registration 12
2.2.4.1 Redirect URI 12
2.2.4.2 Client Keys 12
2.3 Connection to the Protected Resource 13
2.3.1 Requests to the Protected Resource 13
3 Authorization Server Profile 13
3.1 Connections with Clients 13
3.1.1 Grant Types 14
3.1.2 Client Authentication 14
3.1.3 User Approval of the Client's Authorization 14
3.1.4 Discovery 16
3.1.5 PKCE 17
3.1.6 Redirect URIs 18
3.2 JWT Access Tokens 19

3.3 Refresh Tokens 20
3.4 Connections with Protected Resources..21
3.4.1 Introspection 21
3.5 Response to Authorization Requests 21
3.6 Token Lifetimes..22
3.7 Scopes 22
3.8 Protecting Resources 22
3.9 Viewing and Revoking Client Accesses and Tokens 22
3.10 Audit 23
4 Protected Resource Profile 23
4.1 Connections from Clients 23
4.2 Connections to Authorization Servers..24
5 Security Rationale for Profile Requirements..24
6 Security Considerations 27
7 Normative Reference 28
8 Informative Reference 29
Acronyms...31

List of Figures

Figure 1 Example Web Application OAuth Protocol Flow 4
Figure 2 Example Web Application OAuth Protocol Flow using Profile Requirements (Not Exhaustive) 5
Figure 3 Example Native Application OAuth Protocol Flow 7

1 Introduction

This document profiles the OAuth 2.0 web authorization framework [RFC6749] for use in the context of securing web-facing application programming interfaces (APIs), particularly Representational State Transfer (RESTful) APIs. The OAuth 2.0 specifications accommodate a wide range of implementations with varying security and usability considerations, across different types of software clients. The OAuth 2.0 client, authorization server, and protected resource profiles defined in this document serve two purposes:

1. Define a mandatory baseline set of security controls, while maintaining reasonable ease of implementation and functionality.
2. Define objective requirements for use of features that provide stronger security properties but are not yet widely available in OAuth implementations.

This OAuth profile is derived from the International Government Assurance Profile (iGov) for OAuth 2.0 [OpenID-iGov] produced by the OpenID Foundation and has been tailored for use in enterprise environments, as further described in section 1.4. This profile incorporates many recommendations found in the IETF Internet-Draft “OAuth 2.0 Security Best Current Practice” [Lodderstedt].

Readers are expected to be familiar with [RFC6749]. All requirements in that specification apply; this profile document levies additional requirements for the enterprise environment.

Section 5 of this document provides detailed security rationale for the profiling decisions made.

1.1 Requirements Notation and Convention

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

All uses of JSON Web Signature (JWS) and JSON Web Encryption (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 , the terms "Claim Name", "Claim Value", and "JSON Web Token (JWT)" defined by JSON Web Token (JWT) [RFC7519], and the terms defined by OpenID Connect Core 1.0 [OIDC-Core].

1.3 Conformance

This specification defines requirements for the following components:

- OAuth 2.0 clients.
- OAuth 2.0 authorization servers.
- OAuth 2.0 protected resources.

The requirements include details of interaction between these components:

- Client to authorization server.
- Client to protected resource.
- Protected resource to authorization server.

When a profile-compliant component is interacting with other profile-compliant components, in any valid combination, all components MUST implement the requirements as stated in this specification. All interaction with non-profile components is outside the scope of this specification.

A profile-compliant OAuth 2.0 client MUST support and utilize certain features as described in section 2 of this specification.

A profile-compliant OAuth 2.0 authorization server MUST support and utilize certain features as described in section 3 of this specification.

A profile-compliant OAuth 2.0 protected resource MUST support and utilize certain features as described in section 4 of this specification.

1.4 Environment Overview

This profile is intended for use in enterprise environments, not consumer-facing environments. In enterprise environments, users do not "own" their data, the enterprise does. However, the user may have some level of responsibility for ensuring that unauthorized entities do not access data that the user has permission to access. In general, users need to be strongly identified in enterprise environments and not be able to act anonymously when accessing data.

The enterprise is assumed to have a deployed Public Key Infrastructure (PKI). The PKI issues each end user a certificate attesting to the user's identity. The PKI also issues non-person entity (NPE) certificates to clients, protected resources, and authorization servers. As discussed later, the PKI can be leveraged to provide greater assurance than is present in current typical non-enterprise OAuth deployments.

Users typically have authorization attributes associated with them by the enterprise representing what types of data the user is permitted to access or what operations the user is allowed to perform. Clients similarly may have authorization attributes associated with them. However, the specific details of these attributes are out of scope for this profile. Future profiles may attempt to

standardize common attributes seen in enterprise environments. In some cases, it may make sense to include these attributes (or the intersection of the user's attributes and client's attributes when applicable) in OAuth access tokens issued by the authorization server. In other cases, it may make sense to omit these attributes from access tokens, in which case protected resources could present the user's identity and client's identity (as asserted in the access token) to a separate enterprise authorization server to obtain attributes or access control decisions.

1.5 Use Cases

This profile is oriented around two primary use cases: user authorization delegation to a web application, and user authorization delegation to a native application.

This profile is not intended to describe user authentication to a web application / server. OpenID Connect, which builds upon OAuth, is intended for that use case. OpenID Connect is profiled in a separate document.

This use case section is non-normative and is intended to provide examples to set the stage for the rest of the profile document.

1.5.1 User Authorization Delegation to a Web Application

In this use case, a web application requires the ability to access a protected resource on behalf of a user, making use of some subset of the user's privileges. A web application is a capability provided by a web server running on a separate endpoint system than the user.

In a naïve approach, the web application could simply be given the ability to impersonate any user to the protected resource solely by authenticating itself and providing the user's identity. However, this approach does not prove to the protected resource that the user was actually involved in the transaction. Another naïve approach would be for the user to provide authentication credentials (e.g. username/password or PKI private key) to the web application. However, this approach provides the web application with full, unfettered ability to act as if it is the user with any resource.

OAuth enables a safer, limited approach for delegating user authorization to a web application to act on behalf of the user. With OAuth (when used in compliance with this profile), the web application constructs an authorization request and redirects the user's web browser to an authorization server. The user authenticates to the authorization server (or the user's web browser makes use of an existing, authenticated session), and the authorization server redirects the user back to the web application with a one-time-use authorization code. The web application provides the one-time-use authorization code to the authorization server and receives an access token that it then uses to access the protected resource on the user's behalf. The access token is issued based on authentication to the authorization server of both the web application and the user. The access token can be limited to only allow a subset of the user's privileges, although the details of how to represent authorization attributes within access tokens are out of scope of this profile. The access token can be limited to only be valid at a particular protected resource.

In OAuth terminology, the user is known as a “resource owner,” and the web application is known as a “client.” Since web applications have the ability to securely store credentials with which to authenticate themselves to the authorization server, they are known in the OAuth specification as “confidential clients.”

Figure 1 illustrates this use case:

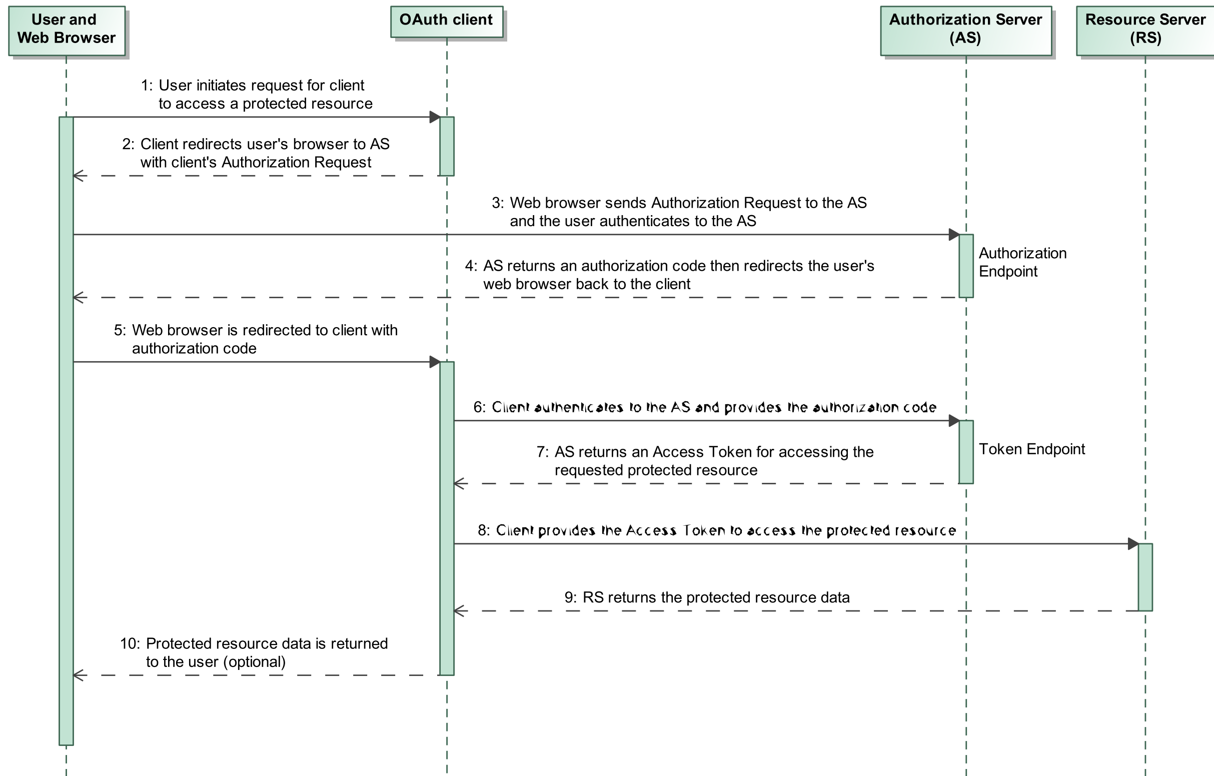

Figure 1 Example Web Application OAuth Protocol Flow

Figure 2 provides a high-level view of this use case including a non-exhaustive overview of this profile’s requirements and recommendations:

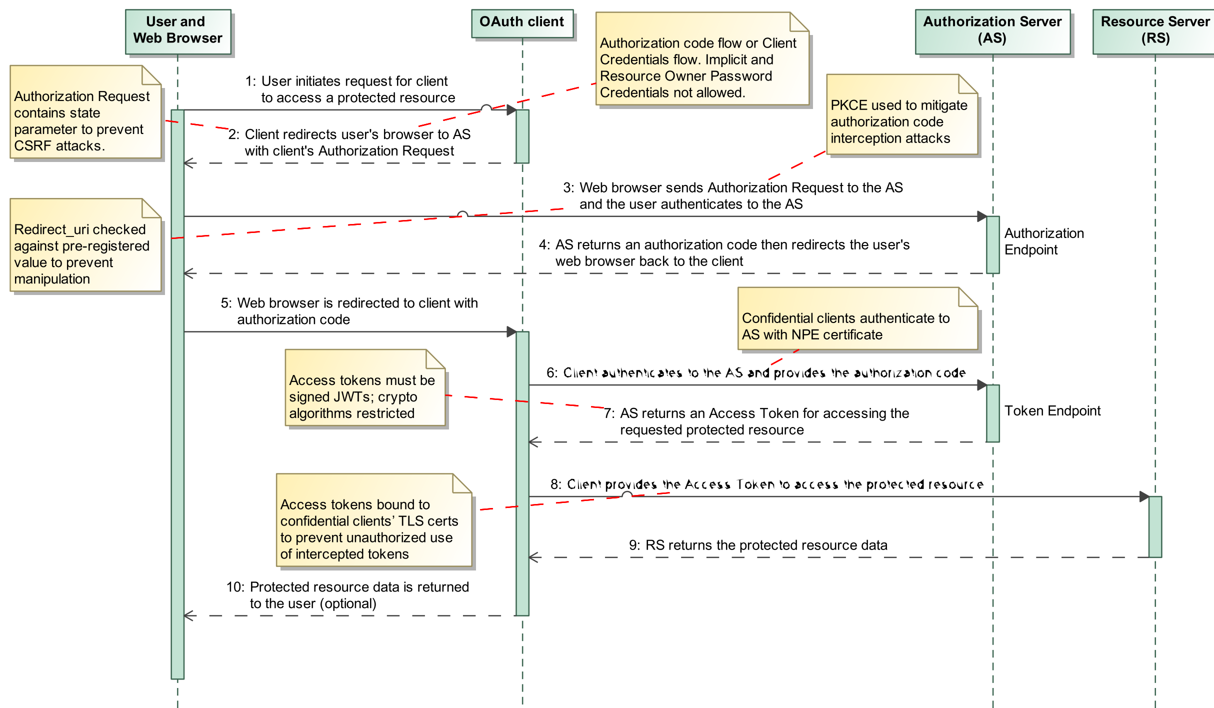

Figure 2 Example Web Application OAuth Protocol Flow using Profile Requirements (Not Exhaustive)

1.5.2 User Authorization Delegation to a Native Application

In this use case, a native application running on the user's endpoint system requires the ability to access a protected resource on behalf of a user, making use of some subset of the user's privileges. For example, an email client may need the ability to access a user's mailbox on an email server.

In a naïve approach, the native application could simply be given the user's authentication credentials (e.g. username/password or private key). However, this approach requires the native application to store those credentials, and if stolen, provides an attacker with full, unfettered ability to act as if he or she is the user with any resource. In the case of a username/password, it also unnecessarily exposes the protected resource to the user's credentials. In addition, this approach limits the flexibility to introduce new authentication methods or perform adaptive authentication (e.g. based on dynamic risk decisions), as those methods would need to be supported by all native applications and all protected resources. For example, TLS client certificate authentication is widely used in some enterprise environments but requiring every app developer to implement client certificate authentication within each app is not feasible.

OAuth enables a safer, limited approach for delegating user authorization to a native application to act on behalf of the user. With OAuth, using the protocol options described in this profile, the native application constructs an authorization request and redirects the user's web browser to an authorization server. The user authenticates to the authorization server through the web browser (or the user's web browser makes use of an existing, authenticated session). Any authentication

method supported by both the web browser and the authorization server can be used, without specific support needed in the application. The authorization server redirects the user back to the native application with a one-time-use authorization code. The native application provides the one-time-use authorization code to the authorization server and receives an access token that it then uses to access the protected resource on the user's behalf. The access token can be limited to only allow a subset of the user's access, and the access token can be limited to only be valid at a particular protected resource. For example, an access token issued to an email client could be valid only for accessing the email server, not other enterprise servers.

In OAuth terminology, the user is known as a “resource owner,” and the native application is known as a “client.” Unlike web applications, native applications typically do not have the ability to securely store credentials with which to authenticate the application itself to the authorization server. The access token is generally issued by the authorization server based on just the user's authentication, not the native application's authentication (the native application provides a client ID, but it typically can be easily captured and spoofed). Applications that do not possess secure credentials with which to authenticate themselves to the authorization server are known in the OAuth specification as “public clients.”

In some cases, rather than use a separate web browser, the native application embeds its own web browser. This approach eliminates the complexity of redirecting the authorization response (containing the one-time-use authorization code) from the web browser back to the native application. However, this approach is generally not appropriate, as it directly exposes the native application to the user's credentials. It may also limit the types of authentication methods that can be used, as the native application may not have functionality for as wide a range of authentication methods as a dedicated web browser.

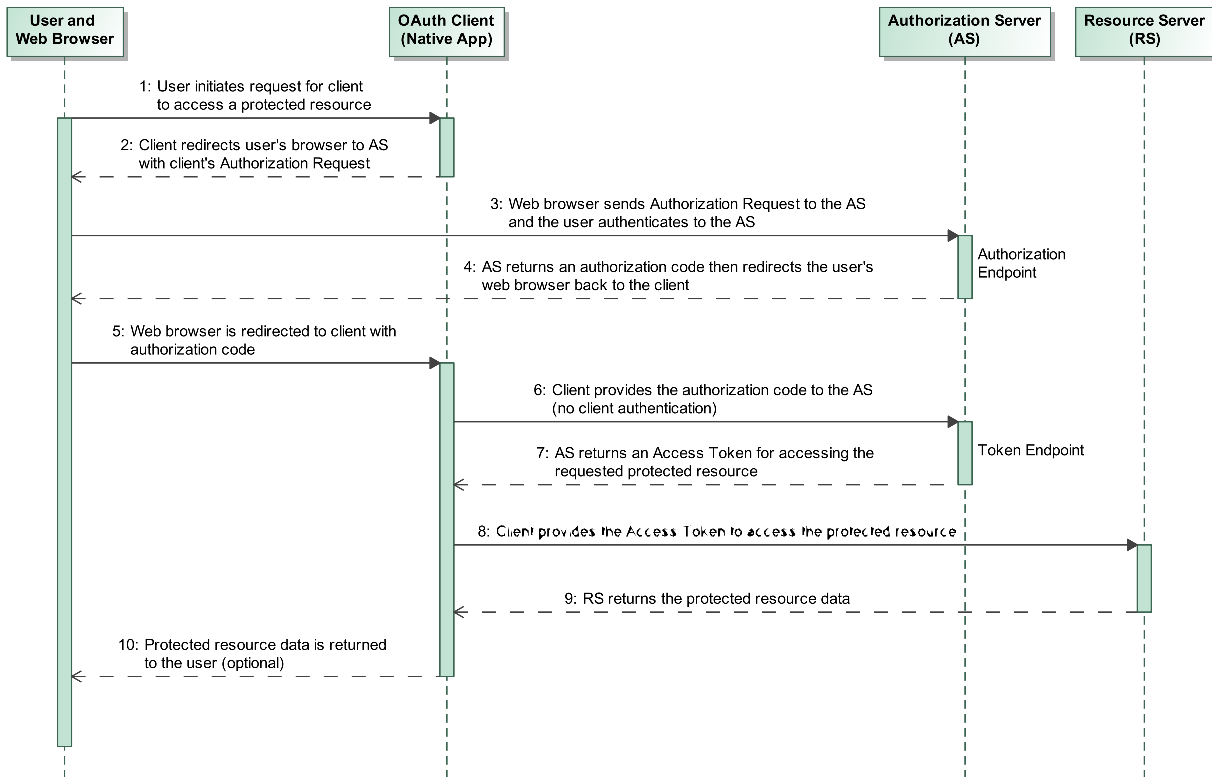

Figure 3 Example Native Application OAuth Protocol Flow

1.5.3 User Authorization Delegation to a Browser-Embedded Client

In this use case, a client application running entirely within the user's web browser requires the ability to access a protected resource on behalf of a user. These applications are typically written in JavaScript and are often referred to as "Single-Page Applications" (SPAs).

At this time, this use case is out of scope for this profile. The IETF Internet-Draft OAuth 2.0 for Browser-Based Apps [Parecki] provides potentially useful details and guidance for this use case, but an examination of its feasibility and security properties would first be necessary.

1.5.4 Token Exchange by Protected Resources

Token exchange is currently out of scope for this profile but will likely be addressed in a future version or additional document. This section provides an initial description of the token exchange use case.

A protected resource (PR1) may need to call a second protected resource (PR2) on behalf of the user in order to satisfy a query received from a client. In some deployments, PR1 could simply use the access token that it received from the client to access PR2. However, this profile requires the access token be sender-constrained and/or audience-constrained, so that would not work. Instead, PR1 must request a new access token from the authorization server that is valid for PR1

to use at PR2 to act on behalf of the user. If PR2 needs to access third resource, PR3, then PR2 must request a new access token, and so on. The IETF Internet-Draft "OAuth 2.0 Token Exchange" [Jones] describes a potential approach for satisfying this need that may be addressed in a future document.

If the protected resources are operated by different organizations, each of which relies on different authorization servers, then the situation is more complex, but can likely still be addressed.

1.6 Global Requirements

This section contains requirements that apply to all of the components described in this profile.

All network connections must use TLS 1.2 or above. Each originator of a TLS connection (the entity acting as a TLS client) must verify the destination's (the entity acting as a TLS server) certificate in accordance with [RFC6125]. Each originator MUST have a capability to limit the certification authorities (CAs) trusted for verifying the destination's PKI certificate. The capability may be provided by the originator itself or by the originator's underlying platform (e.g. operating system on which it is running).

2 Client Profiles

This section profiles the expected OAuth behavior of clients.

2.1 Client Types

This section, and overall profile, distinguishes between two types of clients: confidential clients and public clients.

2.1.1 Confidential Client

The term "confidential client" applies to clients that act on behalf of a particular user and require delegation of that user's authority to access protected resources. Furthermore, these clients are capable of interacting with a web browser application to facilitate the user's interaction with the authorization server. Confidential clients use their own credentials to authenticate themselves to the authorization server, so both the client and the user are authenticated by the authorization server as part of an authorization request.

Typically, confidential clients are front-end web server applications, running on a separate endpoint than the user, as described in Section 1.5.1.

Confidential clients MUST possess their own asymmetric key pair used for authentication to the authorization server. Confidential clients MUST support mutually authenticated TLS (as described in draft-ietf-oauth-mtls) [Campbell] using an X.509v3 certificate [RFC5280] for the client's public key.

2.1.2 Public Client

The term "public client" applies to clients that act on behalf of a particular user and require delegation of that user's authority to access the protected resource. Furthermore, these clients are capable of interacting with a web browser application to facilitate the user's interaction with the authorization endpoint of the authorization server.

Unlike confidential clients, public clients do not use their own credentials to authenticate themselves to the authorization server. Instead, only a client ID (which often can be easily captured) is used. Public clients are typically native applications running on the user's endpoint device, often leading to many identical instances of a piece of software operating in different environments and running simultaneously for different end users. With public clients, generally only the user, not the client, is authenticated by the authorization server as part of an authorization request.

2.2 Connection to the Authorization Server

Confidential and public clients MUST support the OAuth authorization code grant. Confidential clients MAY support the OAuth client credentials grant. Other grant types MUST NOT be used.

OAuth authorization servers provide both an authorization endpoint and a token endpoint. This section profiles connections to these two endpoints from clients. Both the authorization endpoint

and token endpoint are used with the authorization code grant. Only the token endpoint is used with the client credentials grant.

OAuth confidential and public clients do not connect directly to the authorization endpoint. Rather, as described by the OAuth authorization code flow in [RFC6749], the client performs its request by redirecting the user's web browser to the authorization endpoint with appropriate parameters. The user authenticates to the authorization endpoint, and the user's web browser is redirected back to a URI hosted by the client, from which the client obtains an authorization code. The client then presents the authorization code to the authorization server's token endpoint to obtain an access token.

2.2.1 Discovery

Confidential and public clients MAY use the OAuth 2.0 Authorization Server Metadata standard [RFC8414] to retrieve configuration information from the authorization server, including supported options, endpoint URIs, and public keys.

Alternatively, confidential and public clients MAY configure some or all of this information in an out-of-band manner.

2.2.2 Requests to the Authorization Endpoint

Confidential and public clients making a request to the authorization endpoint MUST use an unpredictable value for the state parameter with at least 128 bits of entropy. Confidential and public clients MUST validate the value of the state parameter upon return to the redirect URI and MUST ensure that the state value is securely tied to the user's current session (e.g. by relating the state value to a session identifier issued by the client to the browser).

Confidential and public clients MUST include their full redirect URI in the authorization request. If a confidential or public client provides more than one redirect URI, then it MUST securely tie the authorization request's redirect URI value to the user's current session and ensure that the authorization response is received at the same redirect URI. The client MUST reject the authorization response if it is received at a different URI.

Public clients MUST, and confidential clients SHOULD, in compliance with [RFC7636] using the S256 code challenge method, include the code_challenge parameter and code_challenge_method (set to "S256") in the authorization request. The PKCE code_verifier value MUST contain at least 128 bits of entropy, and it MUST be securely tied to the user's current session (e.g., by relating the code_verifier value to a session identifier issued by the client software to the browser), such that in the client's follow-up request to the token endpoint, the client only presents the code_verifier to the token endpoint that is associated with the same user session.

Confidential and public clients may need to interact with more than one protected resource. If those protected resources are operated by different entities, this may introduce the need for confidential and public clients to interact with more than one authorization server (authorization servers operated by different entities, not a multi-homed approach where a logical authorization

server may have multiple physical instantiations for failover purposes). However, confidential and public clients MUST associate only one logical authorization server with each protected resource. Confidential and public clients MUST use a unique redirect URI for each logical authorization server.

The following is a sample, non-normative response from a client to the end user's browser for the purpose of redirecting the end user to the authorization server's authorization endpoint to perform an authorization request:

```
HTTP/1.2 302 Found
Cache-Control: no-cache
Connection: close
Content-Type: text/plain; charset=UTF-8
Date: Wed, 07 Jan 2015 20:24:15 GMT
Location: https://as.example.com/authorize?client_id=55f9f559-
2496-49d4-b6c3-351a58
6b7484&state=cd567ed4d958042f721a7cdca557c30d&response_type=code
&scope=example_resource&redirect_uri=https%3A%2F%2Fclient%2Eexam
ple%2Ecom%2Fcb
Status: 302 Found
```

This causes the browser to send the following (non-normative) request to the authorization endpoint:

```
GET /authorize?client_id=55f9f559-2496-49d4-b6c3-
351a586b7484&state=cd567ed4d958042f721a7cdca557c30d&response_typ
e=code&scope=example_resource&redirect_uri=
https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb HTTP/1.1
Host: as.example.com
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:31.0)
Gecko/20100101 Firefox/31.0 Iceweasel/31.2.0
Accept:text/html,application/xhtml+xml,application/xml;q=0.9,*/*
;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: https://ehr-va.example.com/portal/signin
Cookie: JSESSIONID=706D5B3A7B3AB3FCE8C6AA7201B8B9CF
Connection: keep-alive
```

2.2.3 Requests to the Token Endpoint

Confidential and public clients connect directly to the token endpoint to retrieve access tokens (and optionally refresh tokens). When the authorization code grant is used, confidential and public clients provide the authorization code they receive as described in the previous section. When the client credentials grant is used, confidential clients do not provide an authorization code (as stated in [RFC6749], public clients cannot use the client credentials grant).

Confidential clients MUST support authentication to the authorization server's token endpoint using mutually authenticated TLS. Public clients MAY support use of mutually authenticated TLS to the authorization server's token endpoint. In the case of public clients, mutually authenticated TLS is not used to authenticate the client to the authorization server, it is used to enable cryptographically binding the access token issued by the authorization server to a private key held by the public client.

Mutually authenticated TLS connections by confidential clients MUST comply with IETF Internet-Draft draft-ietf-oauth-mtls-12 or newer ("OAuth 2.0 Mutual TLS Client Authentication and Certificate Bound Access Tokens") [Campbell]. The self-signed certificate option described in Section 2.2 "Self-Signed Certificate Mutual TLS OAuth Client Authentication Method" MUST NOT be used. Rather, the Section 2.1 "PKI Mutual TLS OAuth Client Authentication Method" MUST be used, where the subject distinguished name (DN) of the client's certificate is registered with the authorization server.

Mutually authenticated TLS connections by public clients, if used, MUST comply with Section 4 of draft-ietf-oauth-mtls-12 or newer.

2.2.4 Client Registration

All clients MUST register with the authorization server.

Client registration MUST be completed by out-of-band configuration; dynamic registration is not supported by this profile.

2.2.4.1 Redirect URI

Clients using the authorization code grant type MUST register their full redirect URIs.

Clients MUST NOT forward values passed back to their redirect URIs to other arbitrary or user-provided URIs (a practice known as an "open redirector").

Android provides a feature called Android App Links [AppLinks], and Apple iOS provides a similar feature called Universal Links [UniversalLinks]. These features provide the ability to enforce a strong binding between a HTTPS URI and a specific mobile app installed on the Android or Apple device. Clients running on the user's endpoint device SHOULD use [AppLinks], [UniversalLinks], or a similar capability enforced by the endpoint device platform to protect their redirect URIs.

2.2.4.2 Client Keys

Confidential clients using mutually authenticated TLS MUST register their certificate's subject DN with the authorization server.

2.3 Connection to the Protected Resource

2.3.1 Requests to the Protected Resource

Clients SHOULD send access tokens to the protected resource in the Authorization header as defined by [RFC6750]. Clients MAY send access tokens using the form-parameter method [RFC6750]. Clients MUST NOT send access tokens using the query-parameter method [RFC6750]. A future version of this profile may remove the form-parameter method option.

Clients SHOULD support mutually authenticated TLS to the protected resource as specified in section 3 "Mutual TLS Client Certificate Bound Access Tokens" of draft-ietf-oauth-mtls-12 [Campbell] or newer. Mutually authenticated TLS will be mandated in a future profile, as it provides strongly desired security properties (further security rationale is provided in section 5) but is not yet widely implemented.

A non-normative example of an OAuth-protected call to a protected resource endpoint, sending the token in the Authorization header, follows:

```
GET /example_resource HTTP/1.1
Authorization: Bearer
eyJhbGciOiJSUzI1NiJ9.eyJleHAiOjE0MTg3MDI0MTIsImF1ZCI6WyJjMWJjOD
RlNC00N2VlLTRiNjQtYmI1Mi01Y2RhNmM4MWY3ODgiXSwiaXNzIjoiaHR0cHM6XC
9cL2lkcC1wLmV4YW1wbGU
uY29tXC8iLCJqdGkiOiJkM2Y3YjQ4Zi1iYzgxLTQwZWMtYTE0MC05NzRhZjc0YzR
kZTMiLCJpYXQiOjE0MTg2
OTg4MTJ9.iHMz_tzZ90_b0QZS-
AXtQtvclZ7M4uDAs1WxCFxpgBfBanolW37X8h1ECrUJexbXMD6rrj_uuWEq
PD738oWRo0rOnoKJAgbF1GhXPAYnN5pZRygWSD1a6RcmN85SxUig0H0e7drmdmRk
PQgbl2wMhu-6h2Oqw-ize
4dKmykN9UX_2drXrooSxpRZqFVYX8PkCvCCBuFy2O-
HPRov_SwtJMk5qjUWMyn2I4Nu2s-R20aCA-7T5dunr0
iWCkLQnVnaXMfA22RlRiU87nl21zappYb1_EHF9ePyq3Q353cDUY7vje8m2kKXYT
gc_bUAYuW-W3SMSw5UlKa
HtSZ6PQICoA
Accept: text/plain, application/json, application/*+json, */*
Host: resourceserver.example.com
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.2.3 (java 1.5)
```

3 Authorization Server Profile

This section details the expected behavior of OAuth Authorization Servers.

3.1 Connections with Clients

3.1.1 Grant Types

The authorization server MUST support the authorization code grant type as described in Section 2 and MAY support the client credentials grant type. The implicit grant type and resource owner password credentials grant types MUST NOT be allowed, and requests attempting to use those grant types MUST be rejected. The authorization server MUST limit each registered client (identified by a client ID) to a single grant type only, since at runtime, a single piece of software will be functioning in only one of the modes described in Section 2. Clients that have multiple modes of operation MUST have a separate client ID for each mode.

Authorization codes issued by the authorization server MUST contain a minimum of 128 bits of entropy and MUST NOT be accepted by the authorization server more than 60 seconds after issuance. The authorization server MUST tie each issued authorization code to a specific client (identified by client ID) and not accept an authorization code if redeemed by a different client. The authorization server MUST NOT accept an authorization code again after it has been redeemed. In a multihomed environment where one logical authorization server is represented by multiple physical instantiations, situations may occur where an authorization code is inadvertently accepted more than once. If this occurs, it MUST be noted in an audit log, any refresh token issued based on the authorization code MUST be revoked, and any access token issued based on the authorization code SHOULD be revoked.

3.1.2 Client Authentication

The authorization server MUST enforce client authentication for confidential clients.

The authorization server MUST support TLS client certificate authentication of confidential clients as specified in draft-ietf-oauth-mtls-12 [Campbell] or newer. The self-signed certificate option described in section 2.2 "Self-Signed Certificate Mutual TLS OAuth Client Authentication Method" MUST NOT be used. Rather, the section 2.1 "PKI Mutual TLS OAuth Client Authentication Method" MUST be used, where the subject distinguished name (DN) of the client's certificate is registered with the authorization server.

The authorization server MAY support mutually authenticated TLS connections from public clients as specified in draft-ietf-oauth-mtls-12 [Campbell] or newer. In the case of public clients, mutually authenticated TLS is not used to authenticate the client to the authorization server, it is used to enable cryptographically binding the access token issued by the authorization server to a private key held by the public client. This requirement is only a MAY because it complicates the TLS configuration of the authorization server, as it would need to be able to validate certificates presented by confidential clients while ignoring validation of certificates presented by public clients. This requirement may be changed to a SHOULD or MUST in a future release of this profile after further lab investigation.

3.1.3 User Approval of the Client's Authorization

The authorization server MUST support the following mechanism for users to authenticate themselves to the authorization server:

- TLS client certificate authentication

The authorization server SHOULD support the following mechanisms for users to authenticate themselves to the authorization server:

- RSA SecurID
- FIDO 2.0 / W3C Web Authentication
- Username and password
- Federated authentication to a user's home organization using OpenID Connect (described below as identity brokering)

The authorization server MAY support other user authentication mechanisms. The authorization server MAY also support the ability to authenticate (and assess security properties of) the user's endpoint device in addition to the user. Such support may be detailed further in a future profile.

The authorization server MUST provide the ability for an administrator to configure which user authentication mechanisms are acceptable.

This profile limits each protected resource to only trusting one authorization server. Since users from multiple organizations may need to access a protected resource, authorization servers typically need to be prepared to authenticate users from those multiple organizations. Several options exist for performing this authentication. If TLS client certificate authentication is used, the authorization server could be configured to trust those organizations' certification authorities (CAs). However, this approach is less practical for authentication methods such as RSA SecurID and username/password. It may also be impractical for FIDO, as it would require the user's FIDO authenticator to be registered with each individual authorization server.

Another approach to authenticate users from other organizations is to perform identity brokering. With identity brokering, the authorization server associated with the protected resource acts as an OpenID Connect Relying Party (RP), delegating authentication to an OpenID Connect Identity Provider (IdP) operated by the user's home organization. The user authenticates to their home Identity Provider, and that IdP asserts to the authorization server that the authentication successfully occurred. If needed, the protected resource's authorization server can obtain attributes about the user from the user's IdP or through some other mechanism. If implemented, identity brokering MUST be performed in accordance with the Enterprise OpenID Connect Profile.

In non-enterprise environments, it is typically desired that the authorization server present the user with the client's authorization request and require the user to explicitly approve the request. However, in this profile, the authorization server MUST provide the ability to disable such functionality. This profile is intended for enterprise environments where individual users do not "own" data. Additionally, this profile requires clients to be approved by the enterprise as part of the client registration process, which provides protection from malicious clients.

If the end user is prompted with an interactive approval page, the authorization server MUST indicate to the user:

- A human readable name of the client
- What kind of access the client is requesting (including scope, target resource, etc.)

3.1.4 Discovery

The authorization server MUST provide an OAuth authorization server metadata endpoint as specified by [RFC8414]. The endpoint MAY be shared with an OpenID Connect discovery endpoint. The endpoint's response MUST contain at least the following fields and MAY contain additional fields:

issuer	The fully qualified issuer URL of the server
authorization_endpoint	The fully qualified URL of the server's authorization endpoint defined by OAuth 2.0
token_endpoint	The fully qualified URL of the server's token endpoint defined by OAuth 2.0
jwks_uri	The fully qualified URI of the server's public key in JWK Set format
introspection_endpoint	The fully qualified URL of the server's introspection endpoint defined by OAuth Token Introspection
revocation_endpoint	(only included if a revocation endpoint exists) The fully qualified URL of the server's revocation endpoint defined by OAuth 2.0 Token Revocation

Note that if the authorization server is also an OpenID Connect Provider, its discovery endpoint must additionally meet the requirements listed in the Enterprise OpenID Connect Profile.

The following non-normative example shows the JSON document found at an authorization server metadata endpoint for an authorization server:

```
{
  "token_endpoint": "https://as.example.com/token",
  "token_endpoint_auth_methods_supported": [
    "tls_client_auth"[1]
  ],
  "jwks_uri": "https://as.example.com/jwk",
  "authorization_endpoint": "https://as.example.com/authorize",
  "introspection_endpoint": "https://as.example.com/introspect",
  "service_documentation": "https://as.example.com/about",
  "response_types_supported": [
    "code"
  ],
  "revocation_endpoint": "https://as.example.com/revoke",
  "grant_types_supported": [
    "authorization_code",
    "client_credentials",
  ],
  "scopes_supported": [
    "profile", "openid", "email", "address", "phone",
"offline_access"
```

[1] Note: The "tls_client_auth" authentication method name has not yet been finalized by the IETF.

```
  ],
  "op_tos_uri": "https://as.example.com/about",
  "issuer": "https://as.example.com/",
  "op_policy_uri": "https://as.example.com/about"
}
```

It is RECOMMENDED that authorization servers provide cache information through HTTP headers and make the cache valid for at least one week.

The authorization server MUST provide its public key (used by the authorization server to sign tokens) in JWK Set format. The key MUST contain the following fields:

kid	The key ID of the key pair used to sign this token
kty	The key type
alg	The default algorithm used for this key

The authorization server MUST provide an RS256 key with a modulus of at least 2048 bits. The authorization server MAY provide additional keys using the following algorithms: RS384, RS512, ES256, ES384, ES512, PS256, PS384, PS512.

The following is a non-normative example of a 2048-bit RSA public key:

```
{
"keys": [

    {
      "alg": "RS256",
      "e": "AQAB",
      "n": "o80vbR0ZfMhjZWfqwPUGNkcIeUcweFyzB2S2T-
hje83IOVct8gVg9FxvHPK1R
eEW3-p7-A8GNcLAuFP_8jPhiL6LyJC3F10aV9KPQFF-
w6Eq6VtpEgYSfzvFegNiPtpMWd7C43
EDwjQ-GrXMVCLrBYxZC-
P1ShyxVBOzeR_5MTC0JGiDTecr_2YT6o_3aE2SIJu4iNPgGh9Mnyx
dBo0Uf0TmrqEIabquXA1-
V8iUihwfI8qjf3EujkYi7gXXelIo4_gipQYNjr4DBNlE0__RI0kD
U-27mb6esswnP2WgHZQPsk779fTcNDBIcYgyLujlcUATEqfCaPDNp00J6AbY6w",
      "kty": "RSA",
      "kid": "rsa1"
    }

] }
```

3.1.5 PKCE

An authorization server MUST support the Proof Key for Code Exchange (PKCE) extension [RFC7636] to the authorization code flow, including support for the S256 code challenge

method. The authorization server MUST NOT allow clients to use the plain code challenge method.

The authorization server MUST require use of PKCE by public clients, rejecting requests to the authorization endpoint from public clients that do not contain a code_challenge. The authorization server MUST be capable of allowing PKCE to be used by confidential clients, and MUST be configurable to require PKCE to be used by either all or specifically designated confidential clients.

The authorization server MUST ensure that if the request to the authorization endpoint contained a code_challenge, then the corresponding request to the token endpoint MUST contain the appropriate code_verifier.

3.1.6 Redirect URIs

The authorization server MUST compare the client's registered redirect URIs with the redirect URI presented during an authorization request using an exact string match and MUST reject requests with invalid or missing redirect URIs.

The authorization server MUST ensure that each redirect URI is one of the following:

- An HTTPS URI referring to a website with Transport Layer Security (TLS) protection or an app installed on the user’s endpoint using [AppLinks], [UniversalLinks], or similar capability
- Hosted on the user's endpoint without involving remote network connectivity (e.g., http://localhost/), however an HTTPS URI protected using [AppLinks], [UniversalLinks], or similar capability is preferred when possible
- Hosted on a client-specific non-remote-protocol URI scheme (e.g., myapp://), however an HTTPS URI protected using [AppLinks], [UniversalLinks], or similar capability is preferred when possible

3.2 Token Issuance Policy

The authorization server MUST be capable of enforcing an authorization policy that must be met in order for tokens to be issued. This policy MUST be customizable by the administrator. This profile does not enforce specific requirements upon capabilities of the authorization policy, but we recommend at least the following attributes be considered:

- Attributes associated with the user's account, such as:
    - Personnel type (e.g. employee vs. contractor)
    - Citizenship
- The user's method(s) of authenticating to the authorization server
- The protected resource being accessed
- Security posture and other properties of the user's endpoint device
- IP address from which the user's endpoint device is connecting

3.3 JWT Access Tokens

The base OAuth specification does not dictate a specific format for access tokens. To facilitate interoperability with protected resources, this profile requires that authorization servers issue cryptographically signed access tokens in the JSON Web Token (JWT) format. The information carried in the JWT is intended to allow a protected resource to verify the authenticity and parse the contents of the token without additional network calls. If the protected resource is not capable of performing these operations, it can make use of token introspection [RFC7662] to request information about the token's authenticity and contents.

An IETF Internet-Draft "OAuth Access Token JWT Profile" [Bertocci], first published after we began work on our profile, proposes a standard access token format. We may revisit this section as the IETF Internet-Draft matures.

The authorization server MUST be capable of including the following claims in issued tokens:

iss	The issuer URL of the server that issued the token.
client_id	The client id of the client to whom this token was issued.
exp	The expiration time (integer number of seconds since from 1970-01-01T00:00:00Z UTC), after which the token MUST be considered invalid.
jti	A unique JWT Token ID value with at least 128 bits of entropy. This value MUST NOT be re-used in another token.
sub	The identifier of the end-user that authorized this client, or in the case of the client credentials grant, the client id of a client acting on its own behalf.
aud	The audience of the token, an array containing the identifier(s) of protected resource(s) for which the token is valid, if this information is known. The aud claim may contain multiple values if the token is valid for multiple protected resources.
cnf	Capability required for requests from confidential clients, optional for requests from public clients. Specified by section 3 of draft-ietf-oauth-mtls (and by section 4 for public clients). Hash of the client's PKI certificate that was presented using TLS mutual authentication between the client and authorization server. This field binds

	the access token to the client's certificate, enabling the protected resource to ensure that only the authorized client can present the access token (over a mutually authenticated TLS connection).

The following claims MUST be included in issued tokens: iss, client_id, exp, sub. One or both of aud and cnf MUST be included.

The authorization server SHOULD be capable of including additional fields in issued tokens, including the following:

nbf	Not before timestamp
iat	Issue timestamp
amr	The user’s authentication method to the AS when the user authorized issuance of this access token.
auth_time	Timestamp of when the user authenticated to the AS in order to authorize issuance of this access token.

The access tokens MUST be signed with JWS. The authorization server MUST support the RS256 signature method for tokens. It MAY support the following additional asymmetric signing methods defined in the IANA JSON Web Signatures and Encryption Algorithms registry: RS384, RS512, ES256, ES384, ES512, PS256, PS384, PS512. The JWS header MUST contain the following field:

kid	The key ID of the key pair used to sign this token

The authorization server MAY encrypt access tokens using JWE. Encrypted access tokens MUST be encrypted using the public key of the protected resource.

3.4 Refresh Tokens

The authorization server MUST require confidential clients to authenticate in order to redeem a refresh token and MUST ensure that the refresh token was issued to the authenticated client.

The authorization server SHOULD provide the capability to bind refresh tokens issued to public clients to a certificate belonging to the client as described in draft-ietf-oauth-mtls Section 4 [Campbell].

The authorization server SHOULD provide the capability to invalidate a refresh token after it is redeemed with the authorization server, preventing the refresh token from being redeemed again.

Mandates on the specific format of the refresh token are out of scope of this profile, as the refresh token is for the internal use of the authorization server, which both generates and consumes the token.

The authorization server MAY sign refresh tokens using JWS and MAY encrypt refresh tokens using JWE. Encrypted refresh tokens MUST be encrypted either using the authorization server's public key or symmetrically encrypted using a secret key held by the authorization server.

3.5 Connections with Protected Resources

3.4.1 Introspection

The authorization server MUST provide a token introspection endpoint. Token introspection [RFC7662] allows a protected resource to query the authorization server for metadata about a token.

The server responds to an introspection request with a JSON object representing the token containing the following fields as defined in the token introspection specification:

active	Boolean value indicating whether or not this token is currently active at this authorization server. Tokens that have been revoked, have expired, or were not issued by this authorization server are considered non-active.
scope	Space-separated list of OAuth 2.0 scope values represented as a single string.
exp	Timestamp of when this token expires (integer number of seconds since from 1970-01- 01T00:00:00Z UTC)
sub	An opaque string that uniquely identifies the user who authorized this token at this authorization server (if applicable).
client_id	An opaque string that uniquely identifies the OAuth 2.0 client that requested this token

The server MAY include additional fields in its token introspection response.

The authorization server MUST require mutual TLS authentication for the introspection endpoint.

A protected resource MAY cache the response from the introspection endpoint for a period of time no greater than half the lifetime of the token. A protected resource MUST NOT accept a token that is not active according to the response from the introspection endpoint.

3.6 Response to Authorization Requests

The following data will be sent as an Authorization Response to the Authorization Code Flow as described above. The authorization response is sent via HTTP redirect to the redirect URI specified in the request.

The following fields MUST be included in the response:

state	The value of the state parameter passed in the authorization request. This value MUST match exactly.
code	The authorization code, a random string issued by the AS to be used in the request to the token endpoint.

3.7 Token Lifetimes

This profile provides RECOMMENDED lifetimes for different types of tokens issued to different types of clients. Specific applications MAY issue tokens with different lifetimes. Any active token MAY be revoked at any time.

For clients using the authorization code grant type, access tokens MUST have a valid lifetime no greater than one hour, and refresh tokens (if issued) SHOULD have a valid lifetime no greater than twenty-four hours.

3.8 Scopes

Scopes define individual pieces of authority that can be requested by clients, granted by users, and enforced by protected resources. Specific scope values will be highly dependent on the specific types of resources being protected in a given interface. OpenID Connect, for example, defines scope values to enable access to different attributes of user profiles.

Authorization servers SHOULD define and document default scope values that will be used if an authorization request does not specify a requested set of scopes.

To facilitate general use across a wide variety of protected resources, authorization servers SHOULD allow for the use of arbitrary scope values at runtime, such as allowing clients or protected resources to use arbitrary scope strings upon registration.

3.9 Protected Resources

Protected resources grant access to clients if they present a valid access token with appropriate authorization claims (e.g. the token's scope claim and potentially other claims conveying detailed authorization information). Access tokens are not required to contain scopes or other claims conveying detailed authorization information. If they do not, the access token asserts the identity of the user (the token's sub claim) and the client (the token's client_id claim), and the protected resource can make use of applicable enterprise authorization services to determine the allowed access.

Protected resources trust the authorization server to authenticate the end user appropriately for the importance, risk, and value level of the protected resource and requested scopes. The authorization server MAY assert different scopes and authorization claims in the access token depending on the method used to authenticate the user.

Authorization servers MAY allow a refresh token issued for multiple scopes to be used to obtain an access token for just a subset of those scopes.

3.10 Viewing and Revoking Client Accesses and Tokens

The authorization server MUST provide an interface for end users to view a list of clients that have been granted access to resources on the user's behalf, and for end users to revoke this access. Revocation MUST revoke any currently valid refresh tokens issued to the client to access

resources on the user's behalf, SHOULD revoke applicable currently valid access tokens, and MUST prevent the client from obtaining new tokens without the authorization server receiving a new authorization request via the user.

Note that revocation of access tokens may not have an immediate impact, as protected resources may not always check the revocation status of access tokens. However, this profile limits access tokens to a lifetime of 60 minutes, and revocation of the corresponding refresh token will prevent the client from obtaining a new access token upon the access token's expiration.

The authorization server SHOULD provide an [RFC7009]-compliant interface for clients to request token revocation.

The authorization server MUST automatically revoke refresh tokens and SHOULD revoke access tokens under the following conditions:

1. User's account has been locked or deleted.
2. User's account credentials under which the tokens were issued have been reported lost or compromised (e.g. password, private key, hardware token, etc.).

3.11 Audit

The authorization server MUST record at least the following activities in an audit log:

1. Issuance of refresh tokens and access tokens to clients.
2. Attempted or successful use of an authorization code more than once.

4 Protected Resource Profile

This section describes the expected behavior of OAuth protected resources (also known as resource servers). The connections with both clients and authorization servers are detailed below.

4.1 Connections from Clients

A protected resource MUST be capable of receiving access tokens passed in the authorization header as described in [RFC6750]. A protected resource MAY also be capable of receiving access tokens passed in the form parameter. A protected resource MUST NOT accept access tokens passed using the query parameter method. A future version of this profile may prohibit using the form parameter.

Protected resources MUST define and document which scopes are required for access to the resource.

Protected resources MUST verify and interpret access tokens using either JWT, token introspection [RFC7662], or a combination of the two.

The protected resource MUST check the aud (audience) claim, if it exists in the token, to ensure that it includes the protected resource's identifier. The protected resource's identifier is the full subject distinguished name (DN) in the protected resource's certificate. The protected resource

MUST ensure that the rights associated with the token are sufficient to grant access to the resource. The protected resource should enforce whatever authorization policy is appropriate for the resource and not depend solely on OAuth.

Each protected resource MUST be limited to only trust tokens from one logical authorization server. A logical authorization server may include multiple physical instantiations of an authorization server for failover purposes operated by a single organization.

Protected resources SHOULD support mutual TLS client certificate bound access tokens as specified in draft-ietf-oauth-mtls (revision 12 or newer) section 3. This support may be mandated in a future version of this profile.

4.2 Connections to Authorization Servers

Protected resources MAY use the OAuth 2.0 Authorization Server Metadata standard [RFC8414] to retrieve configuration information from the authorization server, including supported options, endpoint URIs, and public keys.

Alternatively, protected resources MAY configure some or all of this information in an out-of-band manner.

Protected resources MAY use the OAuth 2.0 Token Introspection protocol [RFC7662] to connect to the authorization server to retrieve information about an access token presented by a client.

5 Security Rationale for Profile Requirements

This section is intended to provide rationale behind this profile's requirements to help the reader understand why certain decisions were made.

This profile requires that clients be registered with authorization servers in an out-of-band manner, rather than allowing dynamic registration of clients. Clients must have some level of trust placed in them, as they are given the capability to access resources on behalf of the user. Phishing attacks have been demonstrated in environments that allow open registration of OAuth clients. For example, in a past incident, an attacker registered a fake "Google Docs" application with Google, and tricked users into granting the application access to their Google-hosted resources [Reddit]. Additionally, unlike in typical consumer-facing environments, this profile (since it is for enterprise use) does not require users to explicitly consent to granting clients access to their resources, making it even more critical that clients be trusted.

This profile requires use of TLS 1.2 or above for all OAuth interactions, as [RFC6749] does not explicitly require that all interactions be protected with TLS. For example, the initial interaction between the user's web browser and an OAuth client could occur over plaintext HTTP, and Fett et al. (section 3.2 of [Fett]) describe how this property could be leveraged to carry out an authorization server mix-up attack.

This profile requires that all TLS connections validate the TLS server's certificate in accordance with [RFC6125] to prevent successful man-in-the-middle attacks. OAuth has many security dependencies on proper authentication of the TLS server, including:

- Retrieval of discovery information, including authorization server endpoint URIs, and the public keys used to verify the signature on tokens issued by authorization servers
- Authenticating the user to the authorization server, particularly if replayable methods such as username/password are used
- Communicating the one-time-use authorization code from the authorization server to the user's web browser, and again from the user's web browser to the client
- Authenticating the client to the authorization server, if the client_secret method is used
- Communicating the access token (and refresh token if applicable) from the authorization server to client
- Communicating the access token from the client to protected resources
- Communicating the refresh token (if applicable) from the client to the authorization server

This profile provides some degree of resilience in case server certificate validation is not sufficient. For example, an attacker may thwart server certificate validation by illegitimately obtaining a valid certificate from a trusted Certification Authority (CA) [Birge-Lee], somehow injecting new trusted Certificate Authority (CA) certificates into endpoints [Goodin], or exploiting unforeseen vulnerabilities in certificate validation routines. Resilience is provided by requiring that clients and protected resources have the capability of limiting the trusted CAs for connections to the authorization server. Additionally, mutually authenticated TLS connections are required by this profile for many network connections. In a mutually authenticated TLS connection, an attacker could potentially still impersonate the TLS server to the TLS client as described above, but would likely be unable to impersonate the TLS client to the TLS server.

This profile requires use of OAuth's authorization code grant, prohibiting use of the implicit grant and resource owner password credentials grant. The client credentials grant may be used as needed for the client's internal operations; it does not provide delegated authorization of a user's access.

The implicit grant is prohibited because it directly exposes the user's web browser to the access token, which may not be ideal, rather than communicating the access token directly from the authorization server to the client. The implicit grant also may provide more opportunity for an attacker to inject unexpected access tokens into the client (e.g. as stated in draft-parecki-oauth-browser-based-apps section 7.8).

The resource owner password credentials grant is prohibited because it directly and unnecessarily exposes the client to the user's password, and because it is not compatible with other authentication methods or with multi-factor authentication (e.g. as stated in draft-parecki-oauth-browser-based-apps section 5).

This profile requires use of the state parameter by clients and authorization servers. The state parameter provides protection from cross-site request forgery (CSRF) attacks. For example, an attacker may perform a request with an authorization endpoint using the attacker's own credentials, obtain a one-time use authorization code, and then perform a CSRF attack to trick a

victim user into injecting the attacker's authorization code into the victim's session with the client, improperly associating the victim's session with the attacker's resources. Proper use of the state parameter prevents this attack.

This profile describes use of Mutual TLS Client Certificate Bound Access Tokens as specified by section 3 of draft-ietf-oauth-mtls-12 [Campbell], mandating its support on authorization servers, and recommending support by confidential clients and protected resources. This approach cryptographically binds the access token to the client that obtained it, requiring the client to authenticate to protected resources using mutually authenticated TLS in order for the protected resource to accept the access token. This approach prevents stolen access tokens (e.g. from the client's storage or from an insufficiently protected network connection) from being used without access to the client's private key. This approach (along with the token's "aud" field) also prevents a protected resource from replaying an access token that a client presented to it into another protected resource.

This profile requires that exact string comparisons be used for redirect URIs. Wildcards are not permitted. Wildcards have led to security issues in the past, for example by allowing attackers to modify redirect_uri values to point to open redirector web pages running on the same domain as the intended redirect_uri. Open redirectors could be abused to redirect the authorization code to an attacker.

This profile requires clients to include their full redirect URI in the authorization request and to check that the redirect URI matches in the authorization response. This profile also requires a unique redirect URI for each authorization server with which the client interacts. Additionally, this profile requires that clients associate each resource server with only one authorization server, and that each resource server only trusts one authorization server. These requirements provide protection from authorization server mix-up attacks. For example, section 3.2 of [Fett] describes an attack where the attacker interferes with the protocol flow to cause confusion about which authorization server the client is interacting with, tricking the client into sending its one-time-use authorization code to the wrong authorization server. Section IV-A of [Fett-2019] describes an attack dependent on a client trusting multiple authorization servers for a particular resource. In this attack, an attacker-controlled authorization server responds to a client's access token request with an access token from a different authorization server, potentially allowing the attacker to bypass the protections of certificate bound access tokens by tricking the legitimate client into performing operations on the attacker's behalf.

This profile requires use of PKCE by public clients and strongly recommends its use by confidential clients. PKCE protects the one-time-use authorization code from use in certain cases if it is intercepted by an attacker. PKCE was originally intended just for public clients, since public clients have no ability to authenticate themselves to the authorization server, and depending on implementation details it may be possible to intercept the one-time-use authorization code on some client platforms (e.g. while being passed from the platform's web browser to the client). PKCE, however, provides security benefits to confidential clients as well. PKCE provides additional resilience from CSRF attacks if the client fails to properly check the state value. It also protects from the attack described by [Sakimura] in which an attacker injects a stolen authorization code into its own session with an OAuth client, attempting to associate the attacker's session with a victim's resources.

This profile prefers confidential clients authenticate themselves to authorization servers using TLS mutual authentication with a client certificate as described in IETF Internet-Draft draft-ietf-oauth-mtls. Traditionally, a shared secret (called a "client_secret" in RFC6749) is used. However, the shared secret approach is not ideal. If an attacker captures the shared secret (e.g. from the client's storage or by intercepting network communication between the client and authorization server), an attacker could impersonate the client in future sessions simply by using the shared secret. The shared secret is likely to be irregularly or never changed. In enterprise environments envisioned by this profile, confidential clients (typically front-end web servers) already possess and use non-person-entity (NPE) PKI certificates. These NPE PKI certificates and the associated private keys are ideal to use to authenticate clients to the authorization server rather than using a shared secret. TLS mutual authentication also provides resilience against man-in-the-middle attacks, as even if an attacker can impersonate the server to the client, an attacker would additionally have to impersonate the client to the server (rather than just pass through an intercepted client_secret value).

Another asymmetric authentication method called "private_key_jwt" is defined by the OpenID Connect Core specification for authentication of the OAuth client to the authorization server. This profile does not allow its use. private_key_jwt has the advantage over client_secret that the private key is not exposed over the network to an attacker. However, it is not as secure as TLS mutual authentication. With private_key_jwt, the client signs an assertion using its private key and attaches the assertion to its request. The assertion is not tied to the content of the client's request, so the client's request is not resilient against man-in-the-middle attacks if the attacker is able to impersonate the server to the client. The assertion could potentially be replayed if the authorization server does not store previously seen "jti" values until the assertion's expiration (a nonce placed in the assertion to prevent replay). Additionally, private_key_jwt uses JSON Web Keys (JWKs) rather than X.509 certificates, so this may require the client to generate and manage another key pair, including ensuring that the authorization server has the client's public key.

Access token injection, described in section 3.6 of [Lodderstedt], is a potential open issue if adversaries can thwart server certificate validation and perform a man-in-the-middle attack on the connection between the client and authorization server. OAuth does not provide a mechanism for clients to determine that the access token received from an authorization server is the expected token, rather it depends on the security of the HTTPS connection between the two entities. A man-in-the-middle could potentially replace an access token sent between authorization server and client with a different access token. The OpenID Foundation’s Financial-grade API Part 2 [OpenID-FAPI2] provides a mechanism to use an OpenID Connect ID token to bind each received access token to a client authorization request. A future version of this profile may adopt that mechanism. If this threat is a concern, it can be addressed by having the client request and verify an ID token in accordance with the Enterprise OpenID Connect Profile.

6 Security Considerations

All transactions MUST be protected in transit by TLS as described in BCP195.

All clients MUST conform to applicable recommendations found in the Security Considerations sections of [RFC6749] and those found in the OAuth 2.0 Threat Model and Security Considerations document.

7 Normative Reference

[AppLinks] Google. “Handling Android App Links”, <https://developer.android.com/training/app-links>.

[OIDC-Core] OpenID Foundation. "OpenID Connect Core 1.0 incorporating errata set 1”, November 2014, <https://openid.net/specs/openid-connect-core-1_0.html>

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC5280] Cooper, D., et al. “Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile”, RFC 5280, DOI 10.17487/RFC5280, May 2008, <http://www.rfc-editor.org/info/rfc5280>.

[RFC6125] Saint-Andre, P. and J. Hodges, "Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509(PKIX) Certificates in the Context of Transport Layer Security (TLS)", RFC 6125, DOI 10.17487/RFC6125, March 2011, <http://www.rfc-editor.org/info/rfc6125>.

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

[RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750,DOI 10.17487/RFC6750, October 2012, <http://www.rfc-editor.org/info/rfc6750>.

[RFC6819] Lodderstedt, T., Ed., McGloin, M., and P. Hunt, “OAuth 2.0 Threat Model and Security Considerations”, RFC 6819, DOI 10.17487/RFC6819, January 2013, <http://www.rfc-editor.org/info/rfc6819>.

[RFC7009] Lodderstedt, T., Ed., Dronia, S., and M. Scurtescu, "Oauth 2.0 Token Revocation", RFC 7009, DOI 10.17487/RFC7009, August 2013, <http://www.rfc-editor.org/info/rfc7009>.

[RFC7519] Jones, M., Bradley, J., and N. Sakimura, “JSON Web Token (JWT)”, RFC 7519, DOI 10.17487/RFC7519, May 2015, http://www.rfc-editor.org/info/rfc7519.

[RFC7523] Jones, M., Campbell, B., and C. Mortimore, “JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants”, RFC7523, DOI 10.17487/RFC7523, May 2015, <http://www.rfc-editor.org/info/rfc7523>.

[RFC7636] Sakimura, N., Ed., “Proof Key for Code Exchange by OAuth Public Clients”, RFC 7636, DOI 10.17487/RFC7636, September 2015, <http://www.rfc-editor.org/info/rfc7636>.

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

[RFC8414] Jones, M., Sakimura, N., and J. Bradley, “OAuth 2.0 Authorization Server Metadata”, RFC 8414, DOI 10.17487/RFC8414, June 2018, <http://www.rfc-editor.org/info/rfc8414>.

[Campbell] Campbell, B., Bradley,J., Sakimura, N., and T. Lodderstedt, “OAuth 2.0 Mutual TLS Client Authentication and Certificate Bound Access Tokens”, August 2019 (Work in Progress), <https://tools.ietf.org/html/draft-ietf-oauth-mtls>.

[Lodderstedt] Lodderstedt, T., Bradley, J., Labunets, A., and D. Frett, “OAuth 2.0 Security Best Current Practice”, July 2019, <https://tools.ietf.org/html/draft-ietf-oauth-security-topicss>.

[Parecki] Parecki, A., and D. Waite, “OAuth 2.0 for Browser-Based Apps”, December 2018, <https://tools.ietf.org/html/draft-parecki-oauth-browser-based-apps-02>.

[UniversalLinks] Apple, “Universal Links for Developers”, <https://developer.apple.com/ios/universal-links/>.

8 Informative Reference

[Bertocci] V. Bertocci, “JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens.”, April 2019 (Work in Progress), < https://tools.ietf.org/html/draft-ietf-oauth-access-token-jwt-00 >

[Birge-Lee] H. Birge-Lee, et al. "Bamboozling Certificate Authorities with BGP." USENIX Security Symposium 2018. https://www.usenix.org/conference/usenixsecurity18/presentation/birge-lee

[Fett] D. Fett, et al. "A Comprehensive Formal Security Analysis of OAuth 2.0." ACM CCS 2016. https://arxiv.org/pdf/1601.01229.pdf

[Fett-2019] D. Fett, et al. “An Extensive Formal Security Analysis of the OpenID Financial-grade API.” 40th IEEE Symposium on Security and Privacy (2019). https://arxiv.org/pdf/1901.11520.pdf

[Goodin] D. Goodin, ArsTechnica. "Sennheiser discloses monumental blunder that cripples HTTPS on PCs and Macs." https://arstechnica.com/information-technology/2018/11/sennheiser-discloses-monumental-blunder-that-cripples-https-on-pcs-and-macs/

[OpenID-iGov] J. Richer, et al. "International Government Assurance Profile (iGov) for OAuth 2.0 – draft 01." https://openid.bitbucket.io/iGov/openid-igov-oauth2-id1.html

[Jones] M. Jones, et al. "OAuth 2.0 Token Exchange." October 2018 (Work in Progress), <https://tools.ietf.org/html/draft-ietf-oauth-token-exchange>

[OpenID-FAPI2] N. Sakimura, et al. "Financial-grade API – Part 2: Read and Write API Security Profile", October 2018, <https://openid.net/specs/openid-financial-api-part-2.html>

[Reddit] Reddit. "New Google Docs phishing scam, almost undetectable."

https://www.reddit.com/r/google/comments/692cr4/new_google_docs_phishing_scam_almost_undetectable/

[RFC8471] A. Popov, et al. "The Token Binding Protocol Version 1.0", RFC8471, October 2018, <https://tools.ietf.org/html/rfc8471>

[Sakimura] N. Sakimura. "OAuth Profile should mandate RFC7636 (PKCE) for code flow." https://bitbucket.org/openid/fapi/issues/11/oauth-profile-should-mandate-rfc7636-pkce

Acronyms

acr	authentication context class reference
amr	authentication methods reference
API	Application programming interface
CA	Certificate authority
CSRF	cross-site request forgery
DN	Distinguished Name
HTTPS	Hypertext Transfer Protocol - Secure
iGov	International Government Assurance Profile
JSON	JavaScript Object Notation
JWA	JSON Web Algorithms
JWE	JSON Web Encryption
JWK	JSON Web Keys
JWS	JSON Web Signature
JWT	JSON Web Token
NPE	Non-person entity
OIDC	OpenID Connect
PKCE	Proof Key for Code Exchange
PoP	Proof-of-Possession
SAML	Security Assertion Markup Language
URL	Uniform Resource Locator
vot	Vector of Trust
vtr	Vectors of Trust Request