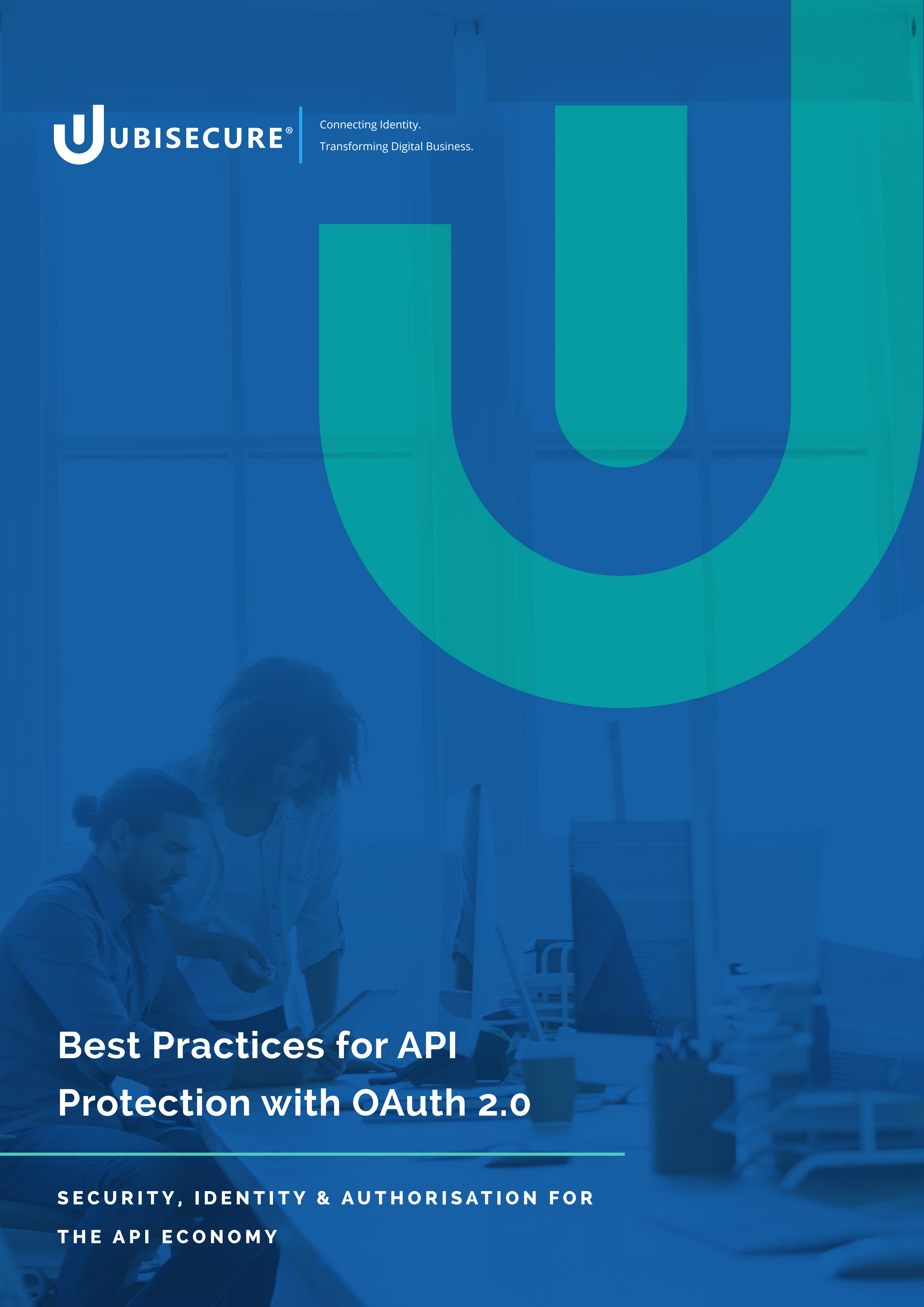
UBISECURE®

Connecting Identity.
Transforming Digital Business.



Best Practices for API Protection with OAuth 2.0

SECURITY, IDENTITY & AUTHORISATION FOR THE API ECONOMY

Contents

1. Introduction

APIs are now the standard entry point to the majority of newly created 'back-end' functionality. These APIs exist to provide not only a standardised, structured way to access the required features or functions, but also to act as 'gatekeepers', ensuring appropriate security, auditing, accounting etc.

Security is always underpinned by identity and as such APIs need to know, if not who is accessing them, what is the context in which they are being accessed.

A variety of techniques of passing either authentication or authorisation data have been used over the years - from additional username/password parameters, to API keys, to full blown OAuth 2.0 based token support.

This whitepaper looks at the background to OAuth 2.0 API protection, , alongside best practices and seeks to dispel some of the complexity and the myths that still surround this approach.

We take the view of the API writer (the 'Resource Server'). Future blogs and white papers will explore the work required in other actors within the eco-system.

2. Terminology

Term	Description
Resource Owner	An entity capable of granting access to a protected resource. In the context of this document, the end user who uses a client to access a protected API.
Resource Server	The server hosting the protected resources. In the context of this document this means the API in question, running on a 'web' server.
Client	An application making protected resource requests on behalf of the resource owner and with its authorisation. 'Application' may be a web application, mobile application, command line script etc. that invokes the protected API.
Authorisation Server	The server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorisation.

Note that these roles are not restricted to a single role per application. In fact, it is common for real world applications to simultaneously act in many roles. For example, an OAuth 2.0 authorisation server often acts in both Authorisation Server and Resource Server roles, and a web application acts in both Client and Resource Server roles.

3. RESOURCE SERVER AND PROTECTED API

This chapter details what a Resource Server needs to implement in order to protect an API with OAuth 2.0.

From the Resource Server's perspective, hosting a protected API is very simple. Each API request from a Client will contain an access token. The Resource Server will validate this access token with the Authorisation Server using a well defined introspection service, provided by the Authorisation Server.

The Resource Owner in collaboration with the Client must generate the required access token, and this is performed using the normal OAuth 2.0 flows for authorisation.

The outline of these two basic steps is shown below:

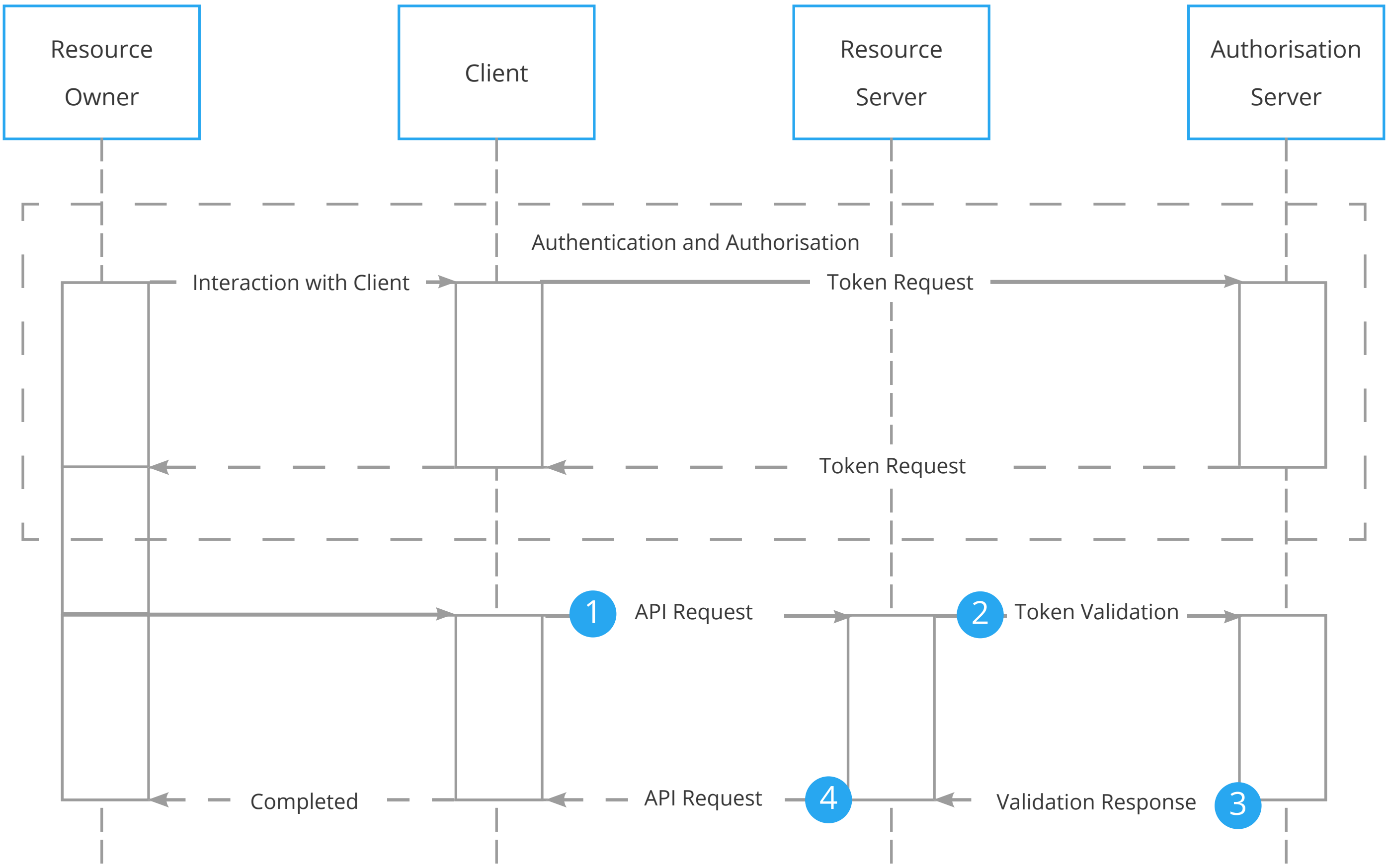


Diagram: OAuth 2.0 and API

3.1 ACCESS TOKEN GENERATION

In the above diagram, the ‘Authentication and Authorisation’ part is simplified, and it is assumed that the Token response message contains an access token for the Client. Different protocols for a Client to get access tokens from an Authorisation server will be presented later in this document.

What is important to understand is that the abstraction of Client, Resource Server and Authorisation Server roles allows the Resource Server to be completely unaware of how a Client gets its access token. This abstraction greatly simplifies the implementation of the Resource Server. Different end users with different clients may all use different authorisation and authentication protocols, and these protocols may evolve with time without any modifications needed to the Resource Server implementation.

3.2 API REQUEST AND TOKEN VALIDATION

— API REQUEST

The syntax of API request is entirely specified by the application and the API. Best practice for OAuth 2.0 is that the Client puts the access token in the HTTP standard Authorisation header using the “Bearer” scheme. OAuth 2.0 puts no other restrictions on parameters, body, content types or other parts of the request.

Below are two API request examples. The first one is “RESTful” with JSON formatted content, and the second is “Web Services” style with SOAP encoding.

RESTful API request	SOAP API request

POST /api
Authorisation: Bearer 179c8216e1a0
Content-Type: application/json
 	
{
  “message”:”hello server”
}
	POST /api
	Authorisation: Bearer 179c8216e1a0
	Content-Type: application/soap+xml
	
	<soap:Envelope>
	  <soap:Body>
	    <message>hello server</message>
	  </soap:Body>
	</soap:Envelope>

— TOKEN VALIDATION

When processing an API request the Resource Server will read the access token from the Authorisation header, and sends the token to the introspection endpoint of the Authorisation Server for validation. The Resource Server must previously have registered an identity with the OAuth 2.0 authorisation server. The introspection request is authenticated with credentials registered for the Resource Server. The example below is using the HTTP Basic authentication protocol.

Request parameters

- token - the access token received with the API request

Introspection request to the Authorisation Server

```
POST /introspection
Authorisation: Basic cmVzb3VyY2VzZXJ2ZXI6c2VjcmV0
Content-Type: application/x-www-form-urlencoded

token=179c8216e1a0
```

— VALIDATION RESPONSE

If the Authorisation Server considers the token valid for the entity making the request, then the validation response will contain a parameter named “active” with boolean value “true”. This is the only mandatory parameter of an introspection response.

Response parameters

- active = true - indicates token is valid for the entity that made the introspection request

Depending on the features and configuration of the OAuth 2.0 Authorisation Server, the response will often contain other parameters and claims including:

- scope - what scopes resource owner granted to client, useful for API request authorisation
- sub - machine readable identifier of resource owner, useful for API request auditing and authorisation
- exp - timestamp indicating time when this token is expected to expire, useful if resource server wants to keep a cache of validation responses
- other parameters and claims such as role, organisation, etc. that may be useful for authorisation or other logic implemented by the API

Introspection response from the Authorisation Server

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

{
  “active”:true,
  “scope”:”api”,
  “sub”:”user@example.com”,
  “exp”:1514764800
}
```

— API RESPONSE

As with the originating API request, the syntax and contents of the API response are entirely specified by the application. OAuth 2.0 puts no restrictions on body, content types or other parts of the response.

RESTful response	SOAP response

`HTTP/1.1 200 OK`
`Content-Type: application/json`
 	
`{`
`  “message”:”hello client”`
`}`
	`HTTP/1.1 200 OK`
	`Content-Type: application/soap+xml`
	
	`<soap:Envelope>`
	`  <soap:Body>`
	`    <message>hello client</message>`
	`  </soap:Body>`
	`</soap:Envelope>`

4. AUTHORISATION AND AUTHENTICATION PROTOCOLS

This chapter presents some protocols the Client may use to get access tokens from an Authorisation server.

4.1 PASSWORD GRANT

Password grant is the simplest protocol for getting an access token from an Authorisation server. However, there are a number of serious security issues with this protocol that need to be considered.

ű Your username and password are disclosed to the Client.
If the Client is a random application downloaded from the app store or a web application hosted by a third party then this is a potential security issue.
This is less of a problem if the Client is provided by the same organisation as the OAuth 2.0 provider, or if the client is, for example, a command line script or other trusted tool that you are running in a trusted environment.

ű Privilege escalation because there is no authorisation.
There is no way to control the scope of access tokens the Client is requesting from the OAuth 2.0 authorisation server.
If the Client is not trustworthy it could easily request access tokens with a much wider scope than what was intended.

ű Your application becomes limited to a username and password authentication mechanism.
Two-factor authentication mechanisms are out of scope when using password grant protocol.

See **OAuth 2.0 Threat Model and Security Considerations** for more details.

The diagram below outlines the flow for Password grant:

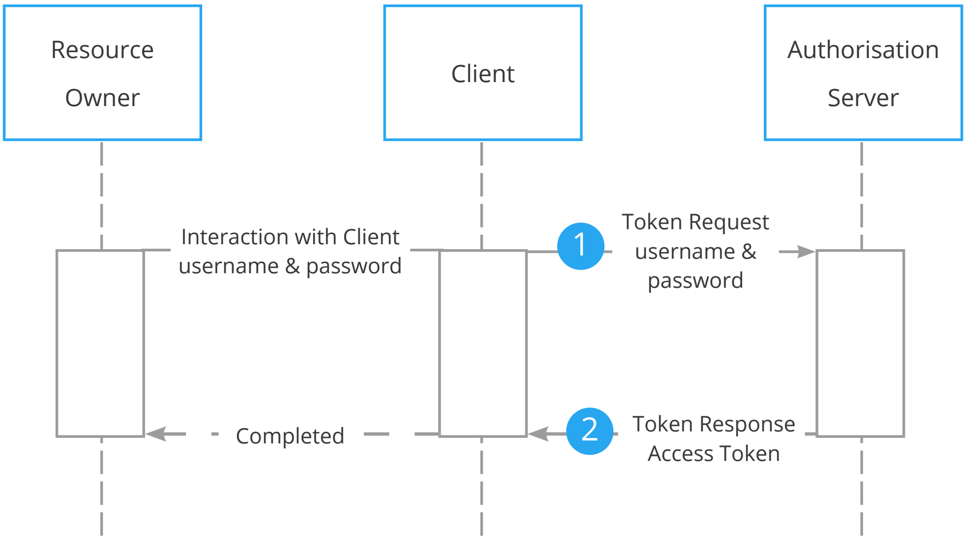

Resource Owner
Client
Authorisation Server
Interaction with Client username & password
1
Token Request username & password
2
Token Response Access Token
Completed

Diagram: OAUth 2.0 resource owner

— TOKEN REQUEST WITH PASSWORD GRANT

Request parameters

- grant_type = password - indicates password grant protocol is being used
- scope - list scopes requested by client
- username - Resource owner username password - Resource owner password

Password grant request

```
POST /token
Authorisation: Basic Y2xpZW50OnNlY3JldA==
Content-Type: application/x-www-form-urlencoded

grant_type=password&scope=api&username=user@example.
com&password=c04ab498d645
```

— TOKEN RESPONSE

Response parameters

- token_type = Bearer - indicates access token is present in response
- access_token - the access token
- expires_in - indicates token expiry time, after which client should assume access token has expired
- scope - list of scopes granted to client

Password grant response

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

{
  “token_type”:”Bearer”,
  “access_token”:”179c8216e1a0”,
  “expires_in”:3600,
  “scope”:”api”
}
```

4.2 AUTHORISATION CODE

The authorisation code grant protocol requires the Resource Owner to use a user agent, such as a web browser.

The authorisation code protocol solves the security issues of the password grant protocol because Resource Owner authentication and authorisation happen within a web browser. No credentials or other confidential information are disclosed to the Client.

Because there is a web browser involved, almost any authentication mechanism is possible.

In addition to web applications, this protocol is also suitable for mobile and desktop applications. A common method in these use cases is where the Client launches the platform's native web browser and starts a very simple localhost web server to capture the Authorisation response.

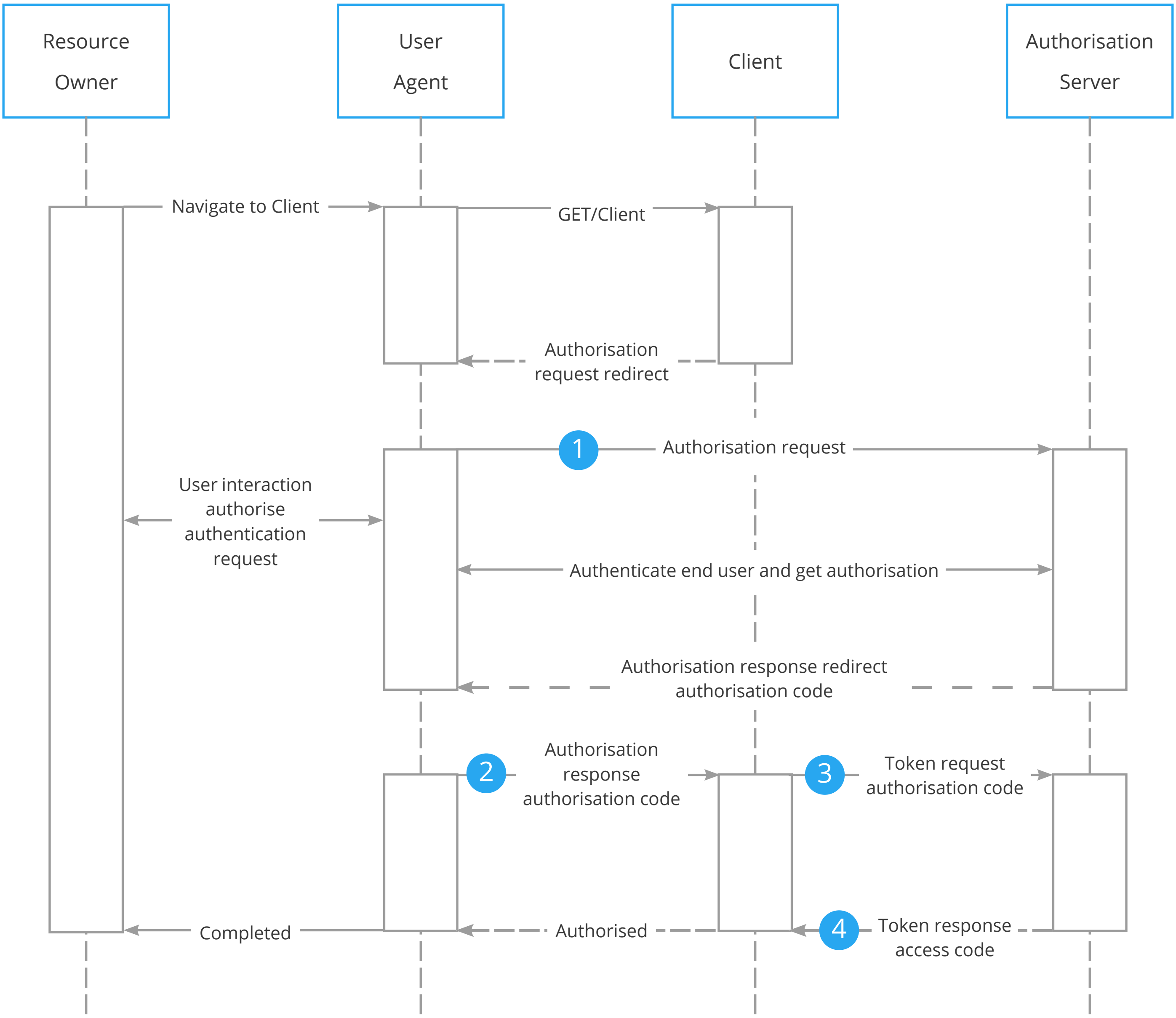

Resource Owner
User Agent
Client
Authorisation Server
Navigate to Client
GET/Client
Authorisation request redirect
1
Authorisation request
User interaction authorise authentication request
Authenticate end user and get authorisation
Authorisation response redirect authorisation code
2
Authorisation response authorisation code
3
Token request authorisation code
Completed
Authorised
4
Token response access code

Diagram: OAuth authorisation code grant

AUTHORISATION REQUEST

Request parameters

- response_type = code
- client_id - client registration identifier
- redirect_uri - redirect uri registered for client
- scope - list scopes requested by client

Authorisation code grant request

```
GET /authorisation?client_id=client&response_type=code&
        redirect_uri=http://localhost:29634/redirect&scope=api
```

AUTHORISATION REDIRECT

Authorisation response is not a direct request from the Authorisation server to the Client. Instead the Authorisation server uses the User agent as intermediary when sending the authorisation code to the Client.

Request parameters

- code - authorisation code received from authorisation server

Authorisation response

```
GET /redirect?code=f2889c08f94e
```

TOKEN REQUEST WITH AUTHORISATION CODE

Request parameters

- grant_type = authorisation_code
- redirect_uri - redirect uri registered for client
- code - authorisation code received from authorisation server

Token request

```
POST /token
Authorisation: Basic Y2xpZW50OnNlY3JldA==
Content-Type: application/x-www-form-urlencoded

grant_type=authorisation_code&code=f2889c08f94e&redirect_
uri=http://localhost:29634/redirect
```

— TOKEN RESPONSE

Response parameters

- token_type = Bearer - indicates access token is present in response
- access_token - the access token
- expires_in - indicates token expiry time, after which client should assume access token has expired
- scope - list of scopes granted to client

Token response

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

{
  “token_type”:”Bearer”,
  “access_token”:”179c8216e1a0”,
  “expires_in”:3600,
  “scope”:”api”
}
```

4.3 CLIENT INITIATED BACKCHANNEL AUTHENTICATION (CIBA)

The CIBA protocol enables the use of an ‘out of band’ authentication mechanism. Often the Authentication device will be software running on a mobile device such as a mobile phone, but other solutions are also possible.

CIBA also solves issues with the password grant protocol because no confidential information is disclosed to the Client.

The authentication device can implement any suitable authentication mechanism. Ranging from simple “click ok” schemes to highly secure mechanisms with biometric verification.

This protocol is suitable for the widest range of applications. In addition to web, mobile and desktop applications, this protocol works with use cases with very limited user interfaces such as call centres, petrol pumps, etc.

The diagram below outlines the CIBA flow:

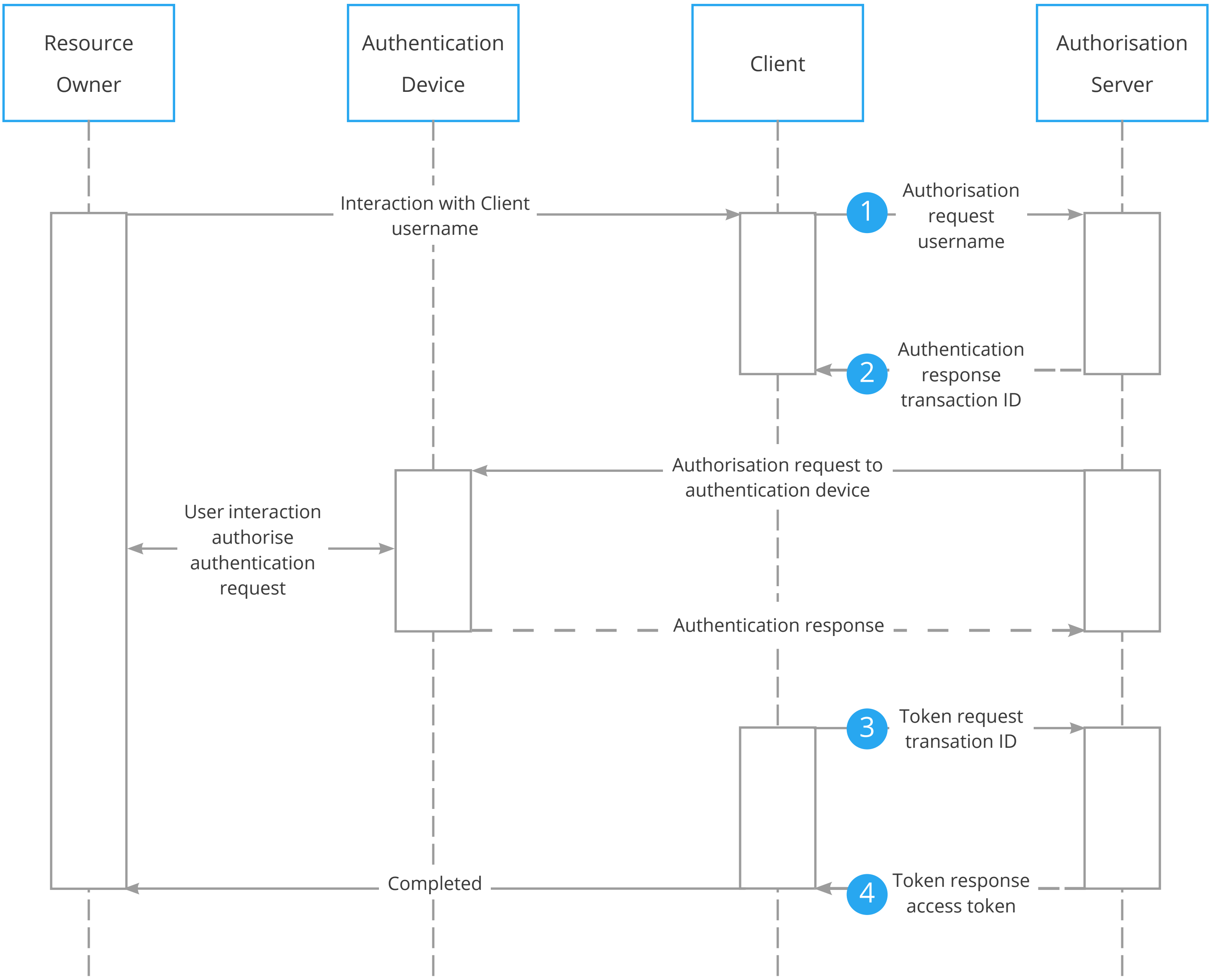

Resource Owner
Authentication Device
Client
Authorisation Server
Interaction with Client username
1
Authorisation request username
2
Authentication response transaction ID
Authorisation request to authentication device
User interaction authorise authentication request
Authentication response
3
Token request transation ID
4
Token response access token
Completed

Diagram: Client Initiated Backchannel Authentication

— AUTHENTICATION REQUEST

Request parameters

- login_hint - username, mobile phone number or other identifier that allows authorisation server reach authentication device of Resource owner
- scope - list scopes requested by client

Authentication request

```
POST /bc-authorize
Authorisation: Basic Y2xpZW50OnNlY3JldA==
Content-Type: application/x-www-form-urlencoded

login_hint=5551234&scope=api
```

AUTHENTICATION RESPONSE

Response parameters

ű auth_req_id - authentication transaction identifier

Authentication response

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

{
  “auth_req_id”:”450ab58cafe5”
}
```

TOKEN REQUEST WITH TRANSACTION ID

Request parameters

ű grant_type = urn:openid:params:modrna:grant type:backchannel_request

ű auth_req_id - authentication transaction identifier

Token request

```
POST /token
Authorisation: Basic Y2xpZW50OnNlY3JldA==
Content-Type: application/x-www-form-urlencoded

grant_type=urn:openid:params:modrna:grant type:backchannel_
request&auth_req_id=450ab58cafe5
```

TOKEN RESPONSE - POLLING

Response parameters

ű error = authorisation_pending - indicates Authorisation server is still waiting for interaction from Resource owner

Token response (failed poll)

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

{
  “error”:”authorisation_pending”
}
```

TOKEN RESPONSE

Response parameters

- token_type = Bearer - indicates access token is present in response
- access_token - the access token
- expires_in - indicates token expiry time, after which client should assume access token has expired
- scope - list of scopes granted to client

Token response (successful)

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

{
  "token_type":"Bearer",
  "access_token":"179c8216e1a0",
  "expires_in":3600,
  "scope":"api"
}
```

4.4 REFRESH TOKEN

The access token allows the client to make requests of the API for a determined length of time. If the Resource Owner wishes to allow the client to continually access for a long time, a security issue exists where the access token, once created, will be usable for that period without further authentication requests. The use of such a long running token would be considered bad practice. To solve this issue, the Authorisation Server can provide not only an access token, but also a refresh token. The access token will have a 'short' expiry time, but the refresh token will have a long expiry time. When the access token expires the client can use the refresh token to generate a new access token, but this re-issuance by the Authorisation Server enables any changes to be included (underlying account changes, authorisation changes etc.).

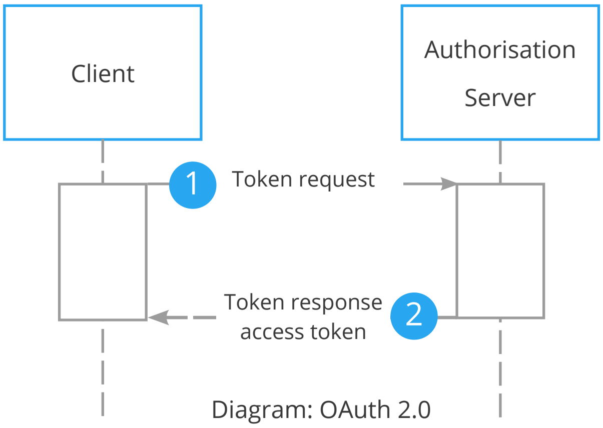


Diagram: OAuth 2.0

— TOKEN REQUEST WITH REFRESH TOKEN

Request parameters

- grant_type = refresh_token
- refresh_token - refresh token is an optional parameter of the Token response

Token request

```
POST /token
Authorisation: Basic Y2xpZW50OnNlY3JldA==
Content-Type: application/x-www-form-urlencoded

grant_type=refresh_token&refresh_token=cd51acb1b04a
```

— TOKEN RESPONSE

Response parameters

- token_type = Bearer - indicates access token is present in response
- access_token - the access token
- expires_in - indicates token expiry time, after which client should assume access token has expired
- scope - list of scopes granted to client

Token response

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

{
  “token_type”:”Bearer”,
  “access_token”:”179c8216e1a0”,
  “expires_in”:3600,
  “scope”:”api”
}
```

5. A COMPARISON OF OAUTH 2.0 AND API KEYS

The pre-cursor to OAuth for APIs has been the use of API keys, and such API key methods are still in common use today. Advocates of API key approaches emphasise simplicity of API keys over the complexity of OAuth 2.0. OAuth 2.0 based solutions bring the advantage of standardisation, and the ability to completely decouple authentication/authorisation from API usage.

The purpose of this comparison is to show that OAuth 2.0 is scalable. It may be as simple as any API keys approach, and it is very easy to build an application that scales from the simple API keys style approach to the full OAuth 2.0 suite.

Area	API Keys	OAuth 2.0 access token
Token differences	Lifetime of API keys often unlimited which makes approach attractive for headless server-to-server approaches	Lifetime of access token often limited - need to use saved credentials or refresh tokens for headless server-to-server approaches
	Ease of use - often there exists an easy to use tool or user interface for generating API keys	Ease of use varies - some protocols for getting access tokens are very simple, others are more complex
Authorisation Server	Equivalent mechanism required, both to generate and initiate (lifetime) key and to validate presented key on API call.	OAuth 2.0 requirement - provides services to generate and validate access tokens
Client	API key is the token and is ready to use for the client	Client must first request an access token using some dynamic mechanism before it can be used
Resource Server	Must validate the API key on presentation	Must validate the access token on presentation

Although there are differences between the two mechanisms, at a high level, and without considering key/token generation, the usage is very similar.

5.1 MIGRATING FROM API KEYS TO OAUTH 2

The work required to migrate an API key based platform to an OAuth 2.0 based platform is quite minimal.

- Put API key in HTTP standard Authorisation header using “Bearer” scheme
- Publish OAuth 2.0 compatible introspection service for validating API keys

This way your API application will be OAuth 2.0 ready.

The existing key management system will have to be exposed to enable access tokens to be created (wrapping existing key meaning). However this can be achieved by a number of off-the-shelf components, such as Ubisecure’s Identity Server.

6. CONCLUSION

Creating a new API server (Resource Server) in an OAuth 2.0 compliant manner is, as has been shown, simple and no more effort than an older API key base manner. More over, using an OAuth 2.0 based authorisation strategy provides significantly more flexibility and security than API keys.

Migrating an existing API service to OAuth 2.0 authorisation is simple and quick and brings many additional security and operational benefits.

7. REFERENCES

https://tools.ietf.org/html/rfc6749

https://tools.ietf.org/html/rfc6750

https://tools.ietf.org/html/rfc7235

https://tools.ietf.org/html/rfc7662

http://openid.net/specs/openid-connect-modrna-client-initiated-backchannel-authentication-1_0.html

https://en.wikipedia.org/wiki/Application_programming_interface_key

API Protection with OAuth 2.0 authored by

Petteri Stenius

Principle Scientist, Ubisecure

About Ubisecure

Ubisecure is a pioneering European b2b and b2c Customer Identity & Access Management (CIAM) software provider and cloud identity services enabler dedicated to helping its customers realise the true potential of digital business. Ubisecure provides a powerful Identity Platform to connect customer digital identities with customer-facing SaaS and enterprise applications in the cloud and on-premise. The platform consists of productised CIAM middleware and API tooling to help connect and enrich strong identity profiles; manage identity usage, authorisation and progressive authentication policies; secure and consolidate identity, privacy and consent data; and streamline identity based workflows and decision delegations. Uniquely, Ubisecure's Identity Platform connects digital services and Identity Providers, such as social networks, mobile networks, banks and governments, to allow Service Providers to use rich, verified identities to create frictionless login, registration and customer engagement while improving privacy and consent around personal data sharing to meet requirements such as GDPR and PSD2.

Ubisecure is accredited by the Global Legal Entity Identifier Foundation (GLEIF) to issue Legal Entity Identifiers (LEI) under its RapidLEI brand, a cloud-based service that automates the LEI lifecycle to deliver LEIs quickly and easily. The company has offices in London and Finland.

UBISECURE®

www.ubisecure.com
sales@ubisecure.com