UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Diploma Thesis

AUDITING AND EXTENDING SECURITY FEATURES OF OAUTH 2.0 FRAMEWORK

Author: *Christinakis Ioannis*

ichristinakis@uth.gr

Supervisor: *Stamoulis Georgios*

Examiners: *Panagiota Tsompanopoulou*

Tsalapata Hariklia

Volos, Greece 2023

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I also declare that the results of the work have not been used to obtain another degree. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism.

The Declarant

Christinakis Ioannis

ΥΠΕΥΘΥΝΗ ΔΗΛΩΣΗ ΠΕΡΙ ΑΚΑΔΗΜΑΪΚΗΣ ΔΕΟΝΤΟΛΟΓΙΑΣ ΚΑΙ ΠΝΕΥΜΑΤΙΚΩΝ ΔΙΚΑΙΩΜΑΤΩΝ

«Με πλήρη επίγνωση των συνεπειών του νόμου περί πνευματικών δικαιωμάτων, δηλώνω ρητά ότι η παρούσα διπλωματική εργασία, καθώς και τα ηλεκτρονικά αρχεία και πηγαίοι κώδικες που αναπτύχθηκαν ή τροποποιήθηκαν στα πλαίσια αυτής της εργασίας, αποτελεί αποκλειστικά προϊόν προσωπικής μου εργασίας, δεν προσβάλλει κάθε μορφής δικαιώματα διανοητικής ιδιοκτησίας, προσωπικότητας και προσωπικών δεδομένων τρίτων, δεν περιέχει έργα/εισφορές τρίτων για τα οποία απαιτείται άδεια των δημιουργών/δικαιούχων και δεν είναι προϊόν μερικής ή ολικής αντιγραφής, οι πηγές δε που χρησιμοποιήθηκαν περιορίζονται στις βιβλιογραφικές αναφορές και μόνον και πληρούν τους κανόνες της επιστημονικής παράθεσης. Τα σημεία όπου έχω χρησιμοποιήσει ιδέες, κείμενο, αρχεία ή/και πηγές άλλων συγγραφέων, αναφέρονται ευδιάκριτα στο κείμενο με την κατάλληλη παραπομπή και η σχετική αναφορά περιλαμβάνεται στο τμήμα των βιβλιογραφικών αναφορών με πλήρη περιγραφή. Αναλαμβάνω πλήρως, ατομικά και προσωπικά, όλες τις νομικές και διοικητικές συνέπειες που δύναται να προκύψουν στην περίπτωση κατά την οποία αποδειχθεί, διαχρονικά, ότι η εργασία αυτή ή τμήμα της δεν μου ανήκει διότι είναι προϊόν λογοκλοπής».

Ο/Η Δηλών/ούσα
Χριστινάκης Γιάννης

ABSTRACT

OAuth 2.0 is a widely adopted authorization framework used in modern web and mobile applications for secure access to protected assets. However, as the framework evolves and new security threats emerge, it becomes crucial to continuously evaluate and enhance its security features.

This thesis aims to address this challenge by conducting a comprehensive audit of the security features of the OAuth 2.0 framework and proposing guidelines for developing a secure OAuth 2.0 implementation and extension features for hardening the infrastructure supporting it.

The analysis utilizes a simple practical threat model as the foundation for understanding vulnerabilities within the framework and their mitigation, which are further explored through an examination of existing OAuth 2.0 implementations and real-world case studies.

Considering these identified threats, a proof-of-concept implementation is developed, tested and reviewed according to a security auditing methodology compiled based on the above.

Finally a set of infrastructure level hardening enhancements and features is proposed in order to strengthen the proof-of-concept system of API services secured using OAuth 2.0.

ΠΕΡΙΛΗΨΗ

Το OAuth 2.0 είναι ένα ευρέως διαδεδομένο πλαίσιο εξουσιοδότησης που χρησιμοποιείται σε σύγχρονες εφαρμογές ιστού και κινητών για ασφαλή πρόσβαση σε προστατευμένα δεδομένα.

Ωστόσο, καθώς το πλαίσιο εξελίσσεται και εμφανίζονται νέες απειλές για την ασφάλεια, καθίσταται ζωτικής σημασίας η συνεχής αξιολόγηση και ενίσχυση των χαρακτηριστικών ασφαλείας του.

Αυτή η εργασία στοχεύει να αντιμετωπίσει αυτήν την πρόκληση διενεργώντας έναν ολοκληρωμένο έλεγχο των χαρακτηριστικών ασφαλείας του πλαισίου OAuth 2.0 και προτείνοντας κατευθυντήριες γραμμές για την ανάπτυξη ασφαλούς υλοποίησης OAuth 2.0 και επεκτάσεις για τη "σκλήρυνση" της υποδομής που το υποστηρίζει.

Η ανάλυση αξιοποιεί ένα απλό, πρακτικό threat model ως βάση για την κατανόηση των τρωτών σημείων εντός του πλαισίου, τα οποία διερευνώνται περαιτέρω μέσω μιας εξέτασης υπαρχουσών υλοποιήσεων OAuth 2.0 και πραγματικών case-studies.

Λαμβάνοντας υπόψη τις απειλές εν λόγω απειλές, αναπτύσσεται, δοκιμάζεται και αναθεωρείται μια proof of concept εφαρμογή, σύμφωνα με μια μεθοδολογία ελέγχου ασφαλείας που συντάχθηκε με βάση τα παραπάνω.

Τέλος, προτείνεται ένα σύνολο βελτιώσεων και τεχνικών σκλήρυνσης σε επίπεδο υποδομής προκειμένου να διασφαλιστεί η ασφάλεια API υπηρεσιών που βασίζονται στο πλαίσιο OAuth 2.0.

Acknowledgements

I would like to express my gratitude towards the colleagues who taught me how to be professional, while maintaining humane values and the university's professors that inspired our curiosity. Their input was invaluable throughout the journey of this thesis.

Moreover, I would like to thank all the significant people, friends and family, for their support and understanding.

Table of Contents

List of Figures

Chapter 1

Introduction

In today's interconnected digital landscape, the secure management of user identities and access to resources is of paramount importance. Identity and Access Management (IAM) frameworks provide the foundation for controlling access to sensitive information and ensuring the integrity and confidentiality of resources. One widely adopted authorization framework within IAM is OAuth 2.0.

1.1 Motivation

While OAuth 2.0 has become a popular choice for granting secure access to protected resources, it is not immune to security vulnerabilities and threats. As the digital landscape evolves and new security risks emerge, it is essential to continuously evaluate and enhance the security features of OAuth 2.0. This investigation aims to compile a comprehensive audit methodology of the framework and propose extensions to improve its overall security, prevent unauthorized access, and mitigate emerging security threats by answering the following question.

- What are the existing security features of the OAuth 2.0 framework ?
- What are the vulnerabilities and potential threats associated with OAuth 2.0 ?
- How can the security features of OAuth 2.0 be enhanced to address known vulnerabilities and emerging threats ?

1.2 Structure

This study is organized into four main chapters, excluding introduction (Chapter 1) and conclusion (Chapter 6).

- **Chapter 2** provides a high level background regarding Identity and Access Management (IAM), Session Management, self-encoded JSON Web Tokens (JWTs) and defines the problem OAuth 2.0 framework was developed to address.
- **Chapter 3** provides a thorough overview of the framework, aiming to identify its key components such as token and client types, roles and protocol flows, as well as to clarify specification terminology.
- **Chapter 4** investigates the potential threats and attack vectors that OAuth 2.0 may be susceptible to, by exploring real-world case studies with the assistance of a practical threat model identifying sensitive assets within the framework's operation in order to consider relevant mitigations and security controls.
- **Chapter 5** presents a robust OAuth 2.0 implementation and extends its architecture, integrating infrastructure components aiming to implement hardening features to assist securing APIs and microservices.

Chapter 2

Background

2.1 Identity and Access Management (IAM)

Identity and Access Management (IAM) is a framework of policies, technologies, and processes that organizations use to manage and control access to digital assets. It ensures that the right entities have the proper access to information, systems, applications, and other resources within an organization's digital ecosystem. [1]

The typical IAM components are the following:

1. Identity Provisioning: The process of creating, modifying, and deleting user accounts and associated access privileges.
2. Authentication: Verifying the identity of users attempting to access resources.
3. Authorization: Granting or denying access permissions to specific resources based on the authenticated user's identity, role, or other attributes.
4. Role-Based Access Control (RBAC): Assigning access permissions based on predefined roles.
5. Single Sign-On (SSO): Allowing users to authenticate once and then access multiple applications or systems without having to re-enter credentials.

6. Identity Federation: Enabling users to access resources across multiple domains or organizations using their existing credentials.
7. Auditing and Compliance: Monitoring and recording user activities, generating audit logs.

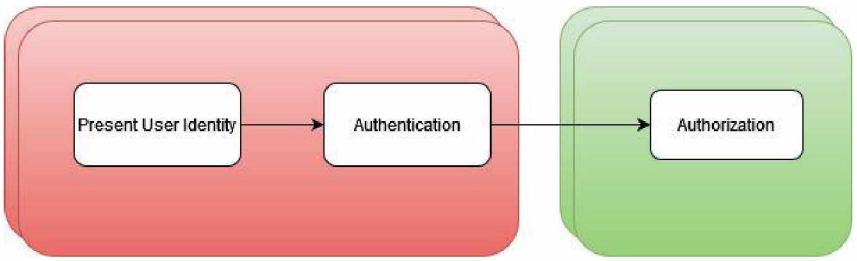

Figure 2.1: IAM operation flow

2.2 Session Management

Session management involves the management and tracking of user interactions and enables maintaining state. In authenticated sessions it also serves the crucial role of linking authentication and authorization within an application. Two common session management mechanisms are cookies and tokens. [2], [3]

Figure 2.2: Session Management

2.2.1 Cookies

Cookie-based authentication involves using HTTP cookies to manage user sessions. When a user logs in, the server generates a session identifier and stores it as a cookie in the user's browser. The browser automatically includes this cookie in subsequent requests, allowing the server to identify and associate requests with the correct session. On user log out the cookie is destroyed both on the client and the server side, thus the session is considered terminated.

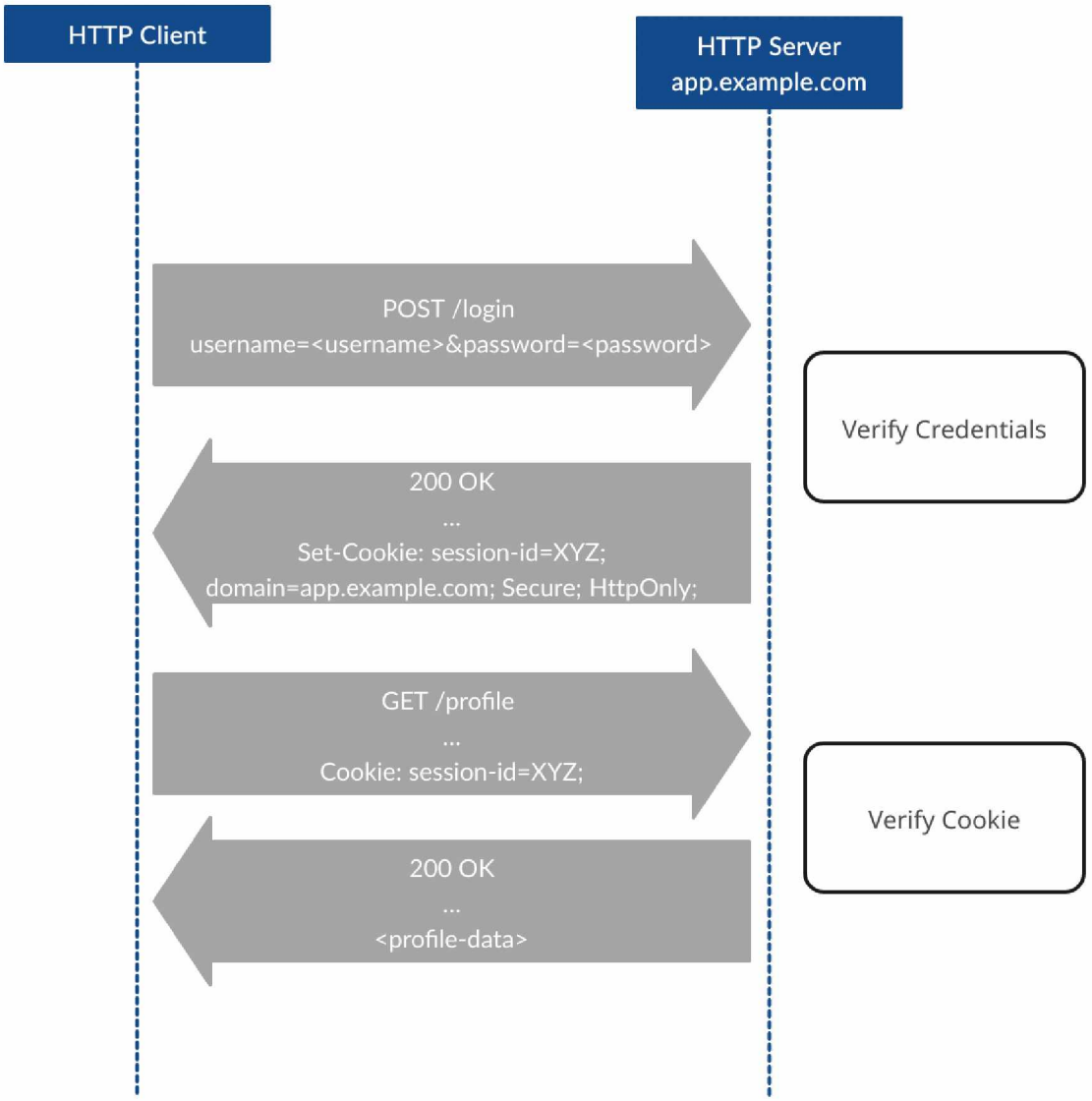

Figure 2.3: HTTP Cookie-based Authentication

While cookie-based authentication remains widely used in web applications that rely on session state and leverage the browser's built-in cookie management capabilities, due to the fact that sessions are stored and managed completely on the server-side, scaling becomes an issue as the user base grows. In addition, cookies are strongly coupled to a single domain and fall short in supporting modern architectures that require cross origin access.

2.2.2 Access Tokens

Decoupling authentication from a single domain is a strong requirement in modern development practices and is the primary factor for token-based authentication's prevalence. Token-based authentication involves issuing a unique token to the client upon successful authentication. The token contains or acts as a reference to information about the user's session or authentication status and is included in subsequent requests as an HTTP header (e.g., Authorization header). [4][5]

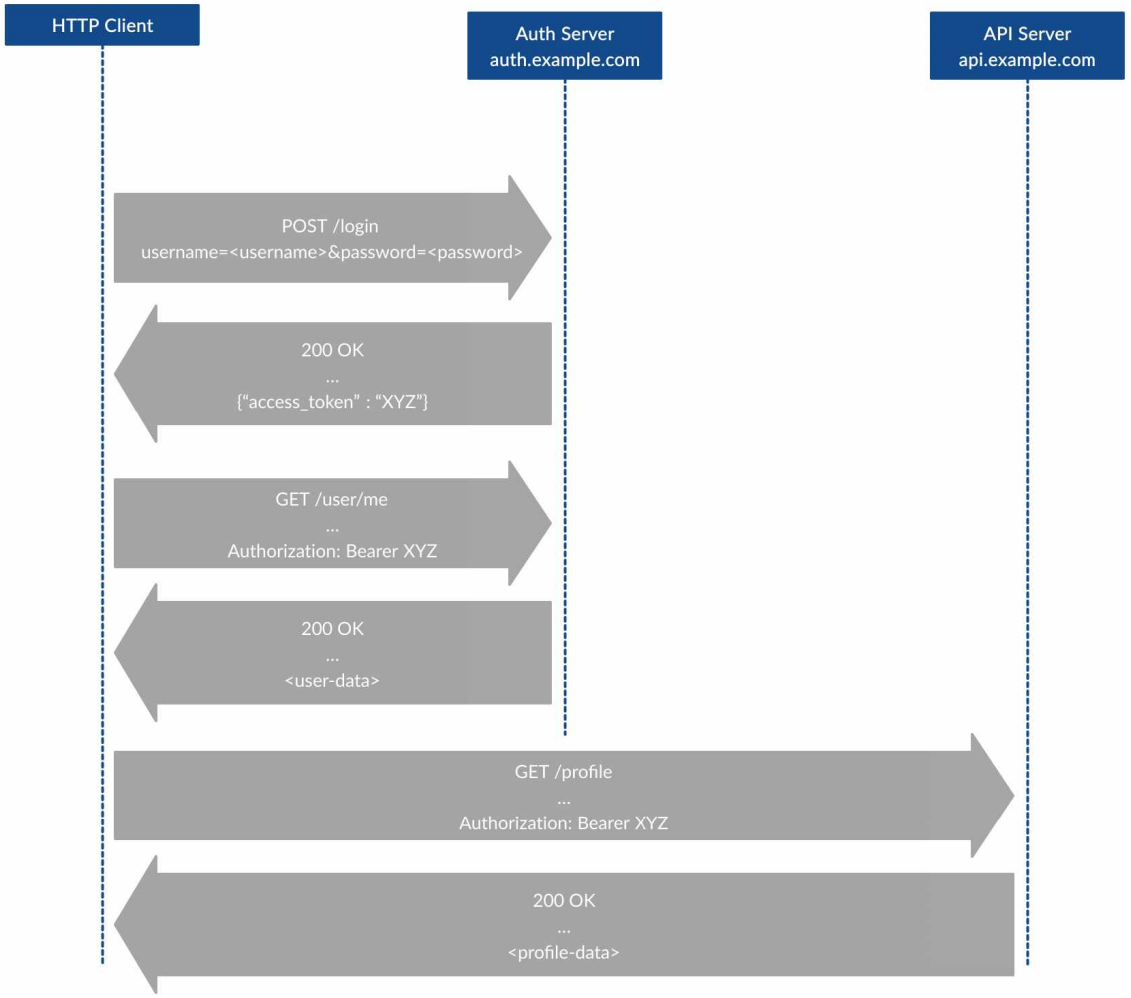

Figure 2.4: HTTP Token-based Authentication

It is tailored for single page and mobile applications, API-based stateless architectures that require cross-platform access and can facilitate implementing SSO capabilities. One of the significant advantages is that it operates in a stateless manner, eliminating the need for the server to store session state, which simplifies server-side implementation and enhances scalability.

However, it does introduce additional complexity in terms of token management. Secure mechanisms for token generation, validation, expiration,

revocation, and token refresh must be established to ensure a secure and seamless user experience.

2.3 JSON Web Tokens

JSON Web Tokens (JWTs) is an open standard [6] that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed, using symmetric or asymmetric cryptography.

A JWT consists of three parts: a header, a payload, and a signature, encoded using Base64 and separated by dots.

1. Header: The header contains information about the type of token and the cryptographic algorithms used for signing and/or encrypting the token and optionally the signing key id.
2. Payload: The payload contains the claims, which are statements about an entity and additional metadata, such as the user's id, roles, permissions, expiration time, or any other custom data. Registered claims are either mandatory or optional. For example *"exp"* claim identifies the token expiration time and is mandatory, while *"jti"* contains the JWT identifier and is optional.
3. Signature: The signature is created by combining the encoded header, payload, and a secret key. It verifies the authenticity of the token and ensures that it has not been tampered with.

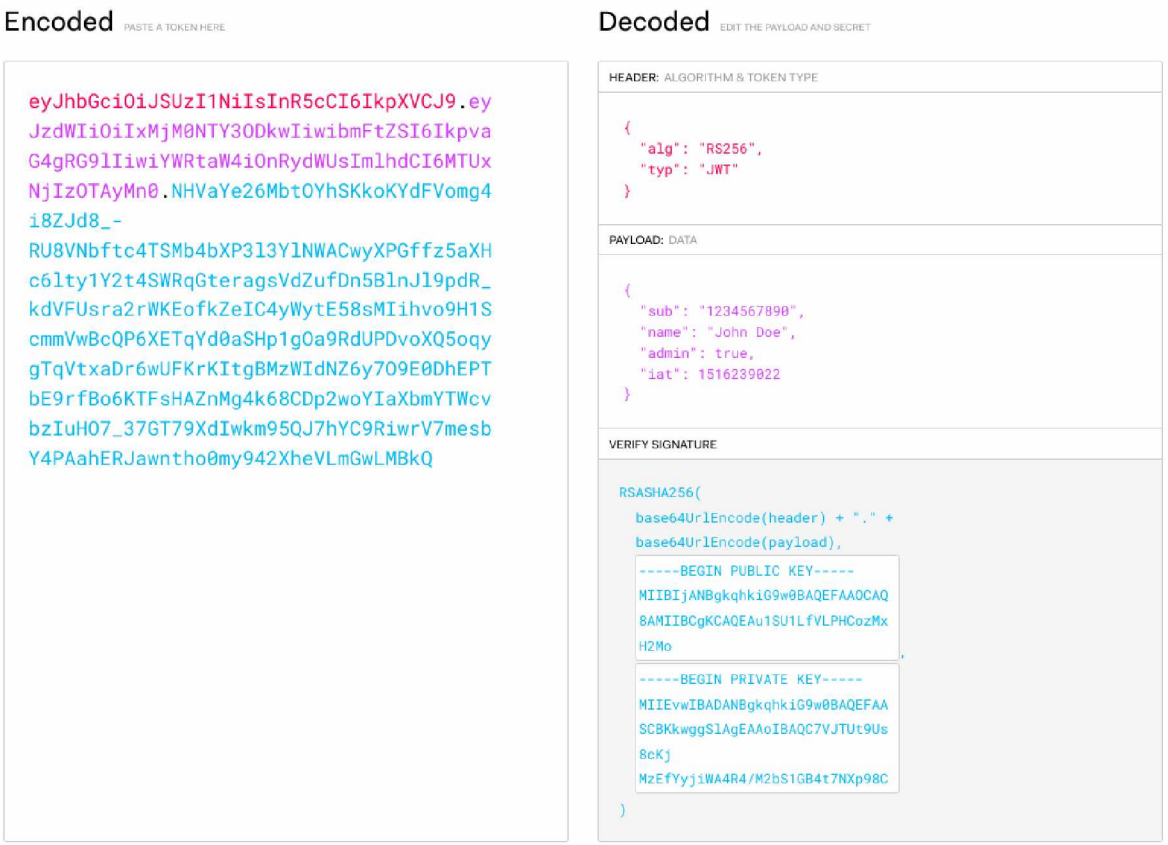

Figure 2.5: JWT token format taken from https://jwt.io/ [7]

JWTs are commonly used for stateless authentication with the key advantage of being easily shared across different systems and platforms. As they are self-encoded, the need for server-side storage or database lookups is limited enabling greater performance and scalability as mentioned in 2.2.2.

JWTs are not managed server side and remain valid until their expiration. This makes auditing and revoking JWTs non-trivial to implement [8].

2.4 The problem with Third-Party Application Authorization

In the pre-OAuth 2.0 era, third-party client application access to user data (documents, social media posts, emails, or contacts) involved users sharing their login credentials (username and password) with the third-party application.

This approach presented several security and privacy concerns and risks, hence is considered obsolete and to be taken as an anti-pattern:

1. Privacy: Sharing login credentials meant providing access to all aspects of the user's account, including potentially sensitive information, rather than providing selective access.
2. Credential Exposure: user credentials were stored by third-party applications increasing the risk of potential data leaks or unauthorized access and misuse.
3. Lack of Control: no means of control over the permissions granted to the third-party application, At the same time revoking access to an already authorized application was challenging and required at the very least a password change.

OAuth 2.0 was developed to address these issues and provide a more secure and controlled way for third-party applications to access user-owned data. It introduces an authorization layer that allows users to grant access to their data selectively, manage authorization preferences and revoke access at any time, while maintaining control and without directly exposing their credentials.

Chapter 3

OAuth 2.0 Framework

3.1 Overview

OAuth 2.0 is an open authorization framework that allows secure access delegation between different systems or applications. It provides a standardized method for granting third-party applications limited access to a resource owner's (typically a user) resources without the need for sharing their credentials with it.

With OAuth 2.0, the user can grant permission to a client application by issuing an access token that represents the level of authorization. This access token is then used by the client application to access the user's resources from the resource server. The client application never sees or handles the user's credentials.

The primary benefits of OAuth 2.0 include improved security by decoupling credentials and enabling controlled resource access, enhanced user experience through seamless authorization, and interoperability across different platforms and services. It offers a flexible and secure way to authorize access to protected resources, such as user data or functionalities, while maintaining user privacy and control.

To demonstrate this with an intuitive example, imagine a visitor comes to a house when the homeowner is not there, and instead of sending the visitor an actual house key, the owner sends them a temporary code to get into a

lockbox that contains the key. OAuth 2.0 operates in a similar manner. In OAuth 2.0, one application sends another application an authorization token to provide user equivalent access, instead of giving out the user's credentials.

3.2 OAuth 2.0 Roles, Client Types and Terms

In this section the key roles, token and client types of OAuth 2.0 are defined and framework terminology is clarified to provide a more practical overview and assist discussing in-depth technical concepts in the following chapters.

3.2.1 The key roles of the OAuth 2.0 framework

- **Authorization Server**: The server responsible for authenticating the user and issuing access tokens.

  For example a trusted Identity Provider such as *Google*'s IdP server which enables federating your Google identity (*Login with Google*) and delegating access to *Google* APIs.

- **Resource Server**: The server hosting the protected resources that the client wants to access on behalf of a user.

  For example is a Mailing service API.

- **Resource Owner**: The entity who owns the protected resources and can grant permission to access them. When the resource owner is a person, it is referred to as an end-user.

Such is, a mailbox owner, the end-user of the Mailing Service.

- **OAuth / Client Application**: a third-party application requesting access to resources on behalf of the resource owner. The term *"client"* does not imply any particular implementation characteristics.

  For example a third-party AI-enabled typing assistant detecting and correcting spelling, grammar, syntactical mistakes in emails and/or documents.

3.2.2 Token Types

- **Access Token**: A *short-lived* token, issued by the authorization server and used by the client application to access protected resources, representing the authorization granted by the user to the client.

  A JWT with *read-only* access to the Mail API, issued by the IdP and used by the typing assistant to access the user's mailbox.

- **Refresh Token**: a token with longer validity, issued by the authorization server and used by the client application to obtain a new access token when the current access token becomes invalid or expires.

  Refresh tokens are meant to provide a frictionless user experience by omitting the need for prompting the user to re-login or re-authorize a client application.

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

3.2.3 OAuth 2.0 Client Types

- **Confidential or Private Client**: A client application able to register a client credentials, maintain their confidentiality and securely authenticate to the authorization server with it, leveraging a server side channel.
- **Public Client**: A client application incapable of maintaining the confidentiality of client credentials, due to the fact it runs in untrusted environments.

  They are typically applications running on user devices or browsers (JavaScript, Android), such as single-page applications, mobile or native apps that pose no server-side channels or storage components.

  Confidential clients are considered to be more secure and trusted and typically are applications with server-side components (Java, Python, C#), which allow them to securely obtain access tokens and store them server-side. Public clients, on the other hand, are considered less secure and untrusted compared to confidential clients.

3.2.4 OAuth Scope

In OAuth 2.0 scope is the term used to specify the level of access or permissions requested by a client application when requesting authorization to access protected resources. The scope also indicates the level of access granted on an issued access token.

Scopes provide a means of fine-grained access control, allowing the resource owner (user) to grant limited and specific permissions to client applications.

The client application specifies the requested scope in the authorization request as an HTTP parameter. However, the scope actually granted on the access token may differ based on the user's consent, which is conveyed in the authorization response body.

For instance, a client application seeks access to a user's email inbox and profile data includes the scopes "read-email" and "read-profile" in the authorization request. However, the user may choose to grant access solely to their profile information, not planning to use the application for email management. Consequently, the issued access token will be scoped exclusively with "read-profile" permissions, allowing the client to access only the resources associated with that specific scope.

3.2.5 OAuth Flows and Authorization Grant

An OAuth flow refers to a series of steps and interactions between the client application, the resource owner , and the authorization server to issue an access token.

These flows define how the resource owner grants authorization to the client application and how it exchanges that authorization, typically through a chain of HTTP redirects, for an access token.

OAuth 2.0 supports multiple flows to cater to different client types and security requirements.

3.3 OAuth Grant Types

OAuth 2.0 framework specifies several grant types, commonly referred to as authorization flows, for different use cases.

For the scope of this thesis, Authorization Code with and without PKCE (Proof Key for Code Exchange) and Implicit flow will be covered in depth, since these are the most widespread and help differentiate between public and confidential clients, while demonstrating the framework's operation and implementation concerns.

Other flows Resource Owner Password flow and Client Credentials flow will be referenced for completion in order to demonstrate the framework's different use cases.

3.3.1 Authorization Code Flow

Authorization Code flow is typically used by confidential clients, such as server-side applications.

It involves the client application redirecting the user to the authorization server's authorization endpoint, providing its client id as an HTTP parameter.

The user is prompted to authenticate and grant consent. If the user provides consent, the authorization server responds with an authorization code and redirects the user to the client's callback endpoint, providing the authorization code as an HTTP parameter.

The client then exchanges this code for an access token and, optionally, a refresh token. [14]

Flow Diagram

A step-by-step graphical representation of Authorization Code flow follows in the diagram below:

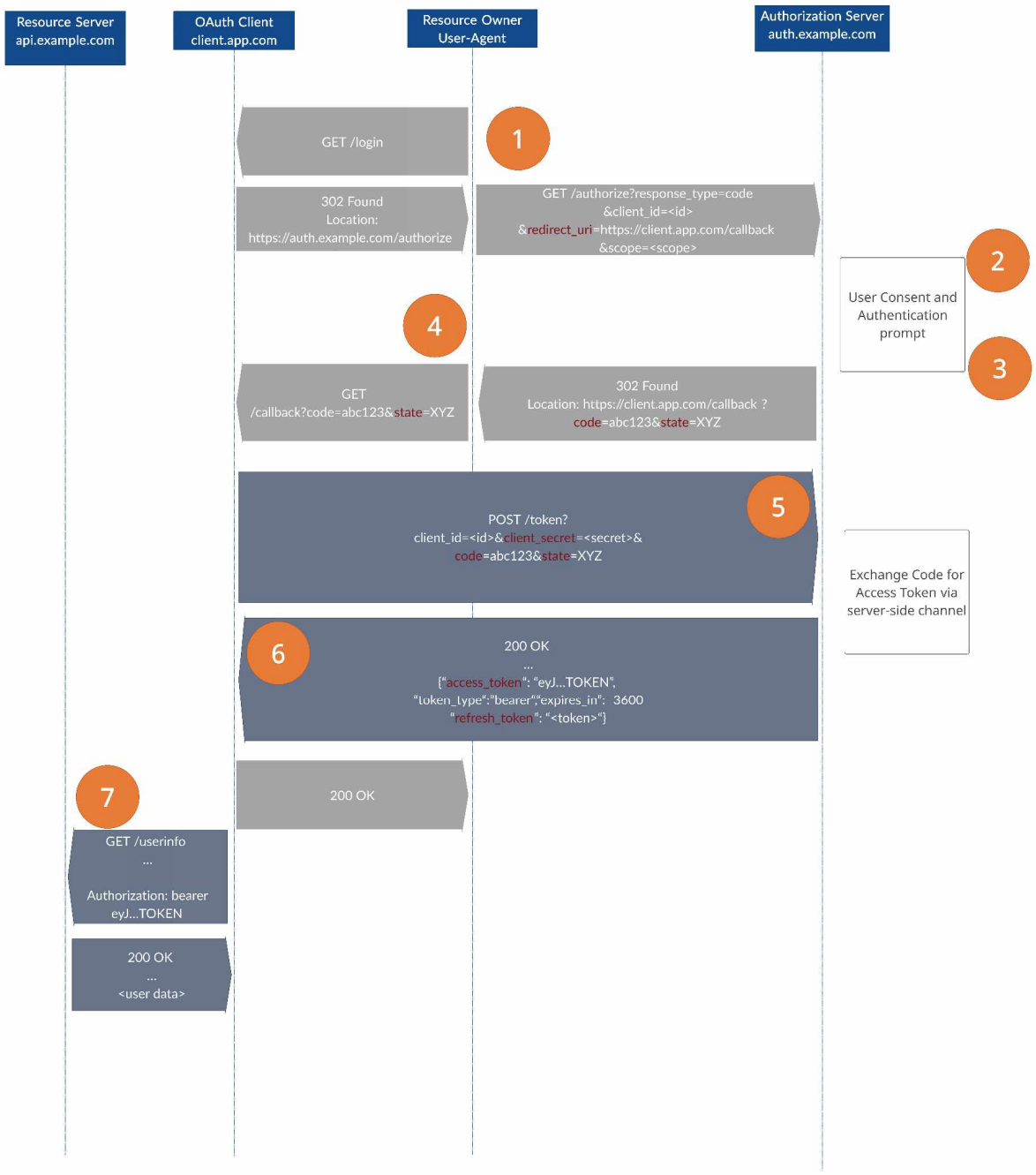

Figure 3.1: Authorization Code Flow. Sensitive parameters with security implications are written in a red font and the dark arrows represent HTTP requests made via confidential, server-side channels. A few parameters are omitted for readability.

1. **Client Initiation**: The client application redirects the user to the authorization server's authorization endpoint, providing its client id in an HTTP parameter. Typically through a user interface element like a "Login with OAuth" button.
2. **User Authentication**: The user is prompted to authenticate with the authorization server. This ensures that the user is aware of the client application and consents to granting it access to their protected resources.
3. **User Consent**: After authentication, the user is presented with a consent screen that explains what permissions the client application is requesting. The user can review the requested permissions and decide whether to grant or deny access.
4. **Authorization Code Request**: If the user grants consent, the authorization server generates an authorization code and sends it back to the client in an HTTP parameter, via a redirect URI specified during the client application registration process.

   The authorization code issued is a temporary credential, strictly tied to the client application and not intended for direct use as an access token.

5. **Authorization Code Exchange**: The client application, using a secure and confidential server-side channel, exchanges the received authorization code with the authorization server for an access token.

   The exchange involves sending a POST request to the authorization server's token endpoint, along with the authorization code, client credentials (client id and client secret), and any additional required parameters.

6. **Access Token Response**: The authorization server verifies the authorization code and client credentials and upon validation, it responds with an access token, which represents the client's authorization to access protected resources on behalf of the user.

   The response may also include other details, such as the token's expiration time, any optional scopes associated with the access token and/or a refresh token.

7. **Accessing Protected Resources**: With the obtained access token, the client application can make requests to protected resources.

3.2.2 Implicit Grant Flow

The Implicit Grant flow is designed for public clients, such as single-page applications (SPAs) or native and mobile apps, where securely storing client credentials is challenging.

This flow does not include client authentication, and relies solely on the presence of the resource owner and the registration of the redirection URI. Since the access token is encoded into the redirection URI as an HTTP or fragment parameter, it may be exposed to the resource owner and other applications residing on the same device or environment (e.g. browser history, browser extensions, JS dependencies, network interception devices, etc). [15]

```
https://client.app.com/callback#<access-token>
```

Flow Diagram

A step-by-step graphical representation of Implicit Grant flow follows in the diagram below:

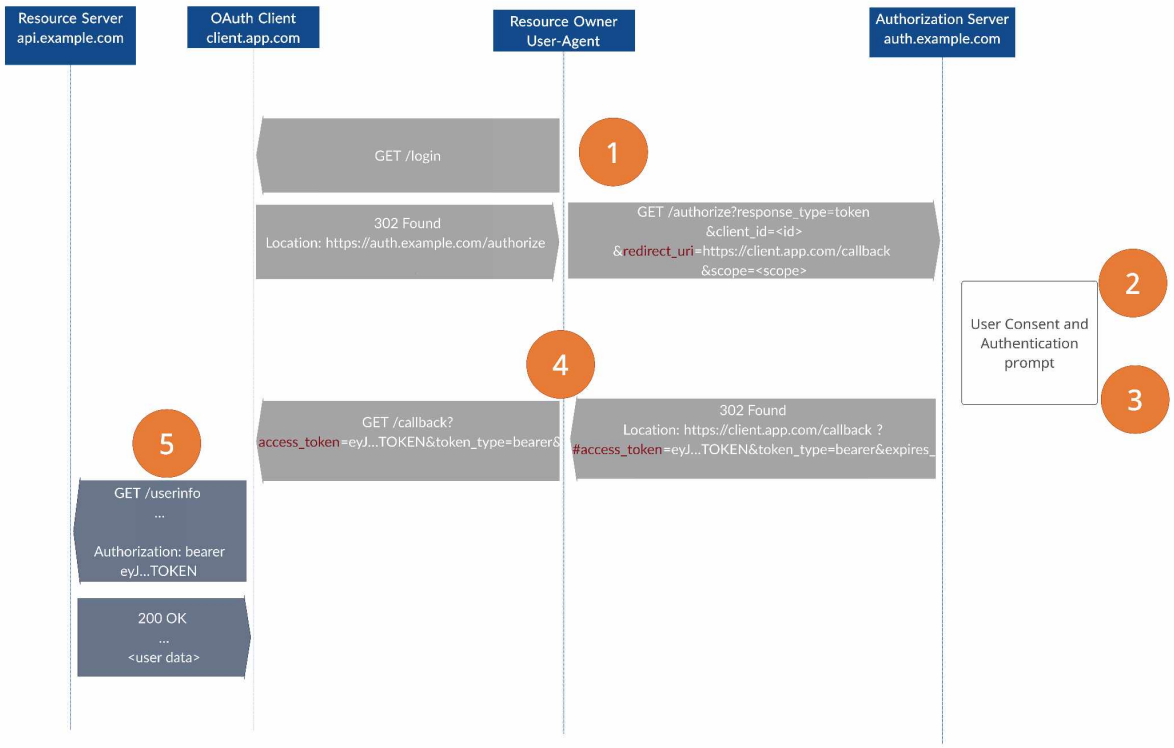

Figure 3.2: Implicit Grant Flow. Sensitive parameters with security implications are written in a red font. The client is public and does not possess a server-side channel. A few parameters are omitted for readability.

1. **Client Initiation**: The client application redirects the user to the authorization server's authorization endpoint, providing its client id in an HTTP parameter. Typically through a user interface element like a "Login with OAuth" button.
2. **User Authentication**: The user is prompted to authenticate with the authorization server. This ensures that the user is aware of the client application and consents to granting it access to their protected resources.
3. **User Consent**: After authentication, the user is presented with a consent screen that explains what permissions the client application is requesting. The user can review the requested permissions and decide whether to grant or deny access.

4. **Access Token Request**: If the user grants consent, the authorization server generates an access token and sends it directly back to the client application.

   Unlike the Authorization Code flow, in which the client makes separate requests for authorization and for an access token, in this flow, the client receives the access directly as a result of the user's authorization and consent, typically via a redirect URL.

5. **Accessing Protected Resources**: With the obtained access token, the client application can make requests to protected resources.

Notes and Reasoning behind Deprecation

Implicit Grant flow has been removed from the latest version of the OAuth 2.1 draft specification [11] and is considered obsolete and insecure, due to lack of support for client authentication, refresh tokens and access token exposure in the redirect URI [16].

Prior to the introduction of PKCE extension (3.2.3) to Authorization Code flow, implementing the Authorization Code flow without a client secret was commonly regarded as a preferable choice over the Implicit grant flow for public clients.

Despite its deprecation it is still widely used, thus relevant and allows demonstrating some key points when assessing OAuth implementations, hence used as reference.

3.2.3 Proof Key for Code Exchange (PKCE)

Proof Key for Code Exchange (PKCE) is an extension to the Authorization Code flow and was introduced to mitigate certain security vulnerabilities associated with public clients by adding an additional step to the Authorization Code flow, ensuring that the authorization code is securely exchanged for an access token by verifying the integrity of the request.

While PKCE enables using Authorization Code flow with public clients, it should not be taken as a replacement for client authentication and does not allow treating public clients as confidential. [17]

Flow Diagram

A step-by-step graphical representation of Authorization Code flow with PKCE follows in the diagram below:

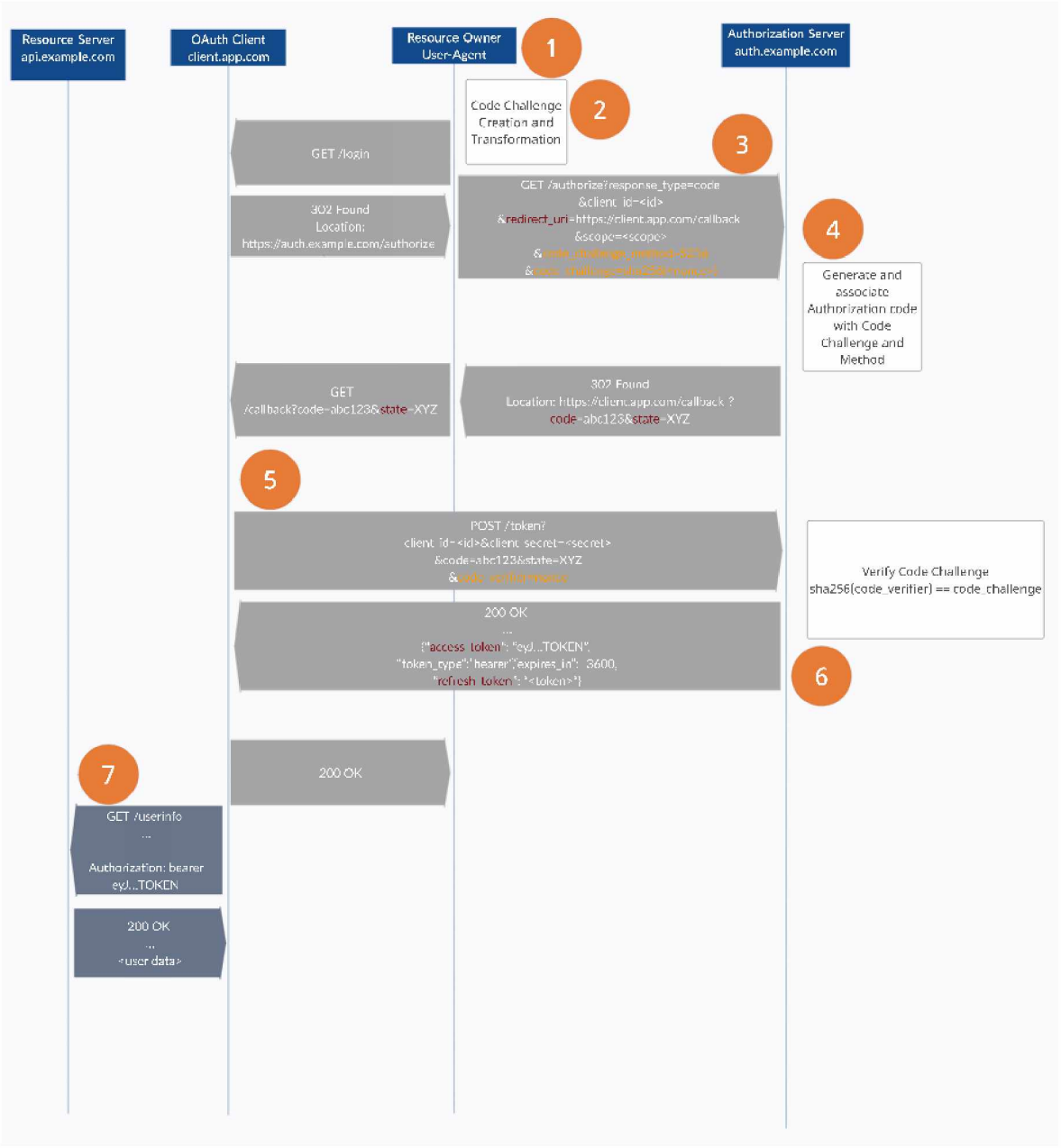

Figure 3.3: Authorization Code flow with PKCE. Sensitive parameters with security implications are written in red font, PKCE specific parameters are written in orange. The client is public and does not possess a server-side channel. A few parameters are omitted for readability.

PKCE specific steps

1. **Code Challenge Creation**: Before initiating the authorization request, the client generates a “code verifier”, a cryptographically random string, “nonce”.
2. **Code Challenge Transformation**: The client then derives a “code challenge” by transforming the “code verifier” using a specified method, such as SHA-256 hashing.

   The “code challenge” is a derived value from the “code verifier”.

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

3. **Authorization Request**: The client initiates the authorization request by redirecting the user's browser to the authorization server's authorization endpoint.

   The request includes the “code challenge” along other necessary parameters.

4. **Authorization Code Exchange**: Upon user authentication and consent, the authorization server generates an authorization code and sends it to the client's specified redirection URI.

   The Authorization server must be able to associate the “code challenge” and method with the generated code. This is done either by storing them in encrypted form within the authorization code or on the server side.

5. **Token Request**: The client application, exchanges the received authorization code with the authorization server for an access token, including the original "code verifier" nonce along the necessary POST request parameters
6. **Access Token Response**: The authorization server validates the authorization code, client credentials, calculates the "code challenge" using the "code verifier" and compares it with the previously associated "code challenge".

   If everything is valid, it responds with an access token.

3.2.4 Other OAuth Grant Types

Client Credentials

Client Credentials flow is designed for machine to machine authentication outside the context of a user and in cases where the client is also the resource owner. Clients applications to directly exchange client credentials for an access token. [9]

Device Code

Device Code flow is designed for use by browserless or input-constrained internet-connected devices (like smart TVs, media consoles, digital picture frames, and printers), where requiring the user to input text in order to authenticate during the authorization flow is impractical. It enables obtaining user authorization to access protected resources by using a user agent on a separate device. [10]

Resource Owner Password

Resource Owner Password flow involves the client directly obtaining the user's username and password. The client then sends these credentials to the authorization server to obtain an access token.

The resource owner credentials are used for a single request and are exchanged for an access token, eliminating the need for the client to store the resource owner credentials for future use.

This flow is omitted in the latest specification draft [11] and is considered obsolete and insecure, yet it is still supported by some implementations for backwards compatibility. [12]

Refresh Token

Refresh Token flow allows a client application to obtain a new access token without requiring the user to re-authenticate. It provides a way to refresh an expired or expiring access token, ensuring that the client application can continue accessing protected resources seamlessly. [12]

Chapter 4

OAuth 2.0 Threat Model and Vulnerabilities

This chapter delves into the critical examination of the simplified threat model associated with OAuth 2.0, focusing specifically on practical threats that are inherent to the framework. The objective is to identify vulnerabilities and abuse paths that have the potential to compromise the security of OAuth 2.0 implementations.

To achieve this, potential threat actors and their capabilities within the operational context are identified. Critical assets and the potential impact of their compromise are also analyzed in order to understand the attack vectors targeting these assets and determine the necessary security controls required to mitigate the risks they pose.

Building upon this theoretical overview of OAuth 2.0 security architecture, case studies from publicly disclosed bug bounty program reports and cases of real-world abuse are examined aiming to demonstrate these attack vectors. These case studies provide tangible examples of how these vulnerabilities manifest in practical implementations, highlighting the urgency and importance of addressing them.

Building a complete threat model for the current state of OAuth 2.0 ecosystem falls beyond the scope of this thesis. The model used in this chapter aims to document theoretically a few realistic, framework specific

attack vectors demonstrated in the last section of the chapter and their mitigation countermeasures in order to underline the necessity and importance of implementing sufficient security controls supporting deployments dependent on OAuth 2.0. [18]

4.1 OAuth 2.0 Threat Actors

A threat actor refers to an entity that possesses the intention, capability, and resources to exploit vulnerabilities or cause harm to a system, organization, or individual. Understanding the motives, methods, and capabilities of threat actors within the context of OAuth is crucial in identifying potential risks and implementing an effective security strategy.

Within the scope of this thesis, the following threat actors have been taken into consideration.

- An external user that may possess an account, attempting to exploit implementation vulnerabilities.
- An agent residing within the network, able to spoof and inspect plaintext network communications (e.g. public WIFI).
- Malicious Client application attempting to trick users into giving authorization consent in order to abuse their level of access.
- Compromised Client application, a legitimate Oauth client, whose credentials have been compromised and abused in order to request credentials on behalf of users who already have authorized it.
- A legitimate user access privileges being abused either intentionally or due to leaked access tokens or compromised account credentials.

4.2 Critical assets

OAuth 2.0 is a standardized and secure mechanism for granting and managing access to protected resources, hence in that sense threat modeling should prioritize securing access to user data and identities.

Within the framework's scope the critical assets granting access to user data and enabling identity theft and abuse are the following:

- Access tokens representing the authorization granted by the user and leaking them is considered equivalent to user data compromise. In the context of this thesis access tokens are considered to be JWTs.
- Refresh Tokens can be exchanged for access tokens, thus considered of equitable importance. Refresh tokens are reference tokens issued by the authorization server and stored server-side by confidential clients.
- Authorization Code can be exchanged for an access token thus considered of equitable importance.
- Client Credentials allow impersonating legitimate applications and can be abused for phishing attacks. Furthermore they can be abused for privilege escalation.

  Consider an IdP that allows customers to develop client applications to use its services and at the same time uses the same approach for internally developed client applications used within the organization by its employees. The internal clients may have an extended scope to access internal resources, not meant to be accessed by end users. Compromising the credentials of an internal client enables accessing sensitive internal scopes.

4.3 OAuth 2.0 Attack Vectors and Mitigations

This section aims to provide an analysis of the common attack vectors associated with the OAuth 2.0 framework and a comprehensive overview of practical and effective mitigations to protect against them. By understanding the attack vectors specific to OAuth 2.0, we can develop robust mitigation strategies to ensure the integrity and confidentiality of user data within OAuth-based systems.

4.3.1 Improper Token Management and Validation

Improper token management and validation in OAuth 2.0 refers to vulnerabilities and weaknesses related to the handling and verification of access tokens and refresh tokens issued for OAuth users.

Most notably such issues have to do with:

- Lack of Transport Layer Security (TLS): TLS should be enforced for all OAuth interactions and all requests containing access tokens, initiated by the client after authorization should be done over HTTPS.
- Token expiration: Access and refresh tokens should have a limited lifespan and should expire after a specified time frame.
- Token integrity validation: prior to accepting a token, its integrity, expiration and signature should be verified to ensure forged, expired or tampered tokens are not accepted.
- Token scope validation: the access scope the token was authorized with should also be enforced, preventing scope extension attempts aiming to access resources beyond the user's consent.

- Token Storage: tokens should be securely stored, both on the server and on the client side. Enforce strong encryption and appropriate access controls to protect tokens from unauthorized access. This is a non-trivial problem for public clients.
- Refresh token must be one-time use and bind to a specific client.

4.3.2 Insufficient Redirect URI Validation

Vulnerability

Insufficient redirect URI validation is a security vulnerability that can affect implementations of OAuth 2.0 framework. It takes advantage of inadequate validation or lack of validation of the redirect_uri parameter during the authorization process.

The redirect_uri is a critical parameter used to redirect the user back to the client application after successful authorization. It is specified by the client application and must be pre-registered with the authorization server.

Abusing vulnerabilities in the validation of the redirect URI to manipulate the flow can lead to various security risks, such as: [19]

- Open Redirect: craft a malicious redirect URI that appears legitimate but redirects the user to a malicious website. This can trick the user into providing sensitive information or performing unintended actions.
- Token Leakage: If the redirect URI is not properly validated, it is possible to intercept the authorization code or access token by registering a malicious redirect URI. This enables gaining unauthorized access to the user's account or sensitive data.

Case Study: Slack redirect uri validation bypass

Slack's authorization server was vulnerable to redirect URI validation bypass. It was possible to redirect a client to an attacker controlled domain by adding a suffix to the legitimate registered redirect url. [20]

```
redirect_uri = client.app.com

bypass = client.app.com.attacker.com
```

The payload that would redirect the client to the malicious domain exploiting the vulnerability:

```
https://slack.com/oauth/authorize?client_id=<id>&redirect_uri=http://client.app.com.attacker.com
```

Other notable redirect uri bypass techniques:

open redirect:

```
https://client.app.com/callback?redirectUrl=attacker.com
```

path traversal with open redirect:

```
https://client.app.com/callback/../redirect/?redirectUrl=attacker.com
```

uri parser logic abuse:

```
https://client.app.com/&@foo.attacker.com#@bar.attacker.com
```

Mitigations

To mitigate risks posed by Insufficient Redirect URI Validation, the following countermeasures must be implemented [21]:

- Strict Redirect URI Validation: The authorization server must ensure that the requested and the registered for the client redirect URIs are equal.

The complexity of implementing and managing pattern matching correctly is a high probable root cause for security issues, therefore to simplify the required logic the authorization server should perform exact string matching ensuring the requested redirect URI (scheme, host and path) matches the registered one before redirecting to the callback endpoint.

- Whitelisting Trusted Redirect URIs: The authorization server can maintain a whitelist of trusted redirect URIs that have been explicitly registered by client applications.

Only requests with redirect URIs matching an entry in the whitelist associated with the requesting client should be considered valid.

- Servers on which callbacks are hosted must not expose open redirectors.

4.3.3 Authorization Code injection

Vulnerability

Authorization Code Injection is a security vulnerability that can affect implementations of OAuth 2.0 protocol. This attack takes advantage of vulnerabilities in the handling of the authorization code, temporary code issued when the user grants authorization, which can be exchanged for an access token.

There are multiple causes of authorization code leakage such as:

- exploitation of vulnerabilities like Insufficient Redirect URI Validation or XSS
- leakage through caches or browser history
- authorization response interception

In an Authorization Code Injection scenario, an authorization code that has been compromised is injected in an authorization flow initiated by a malicious user aiming to gain access to the user's account via the client application in the case of confidential clients or directly obtain an access token in the case of public clients. [22]

Mitigations

To mitigate the risks posed by Authorization Code injection, the following countermeasures must be implemented [9]:

- PKCE [3.2.3] was initially developed for mitigating Authorization Code injection. When the attacker attempts to inject an authorization code, the check of the code verifier fails: the client uses its correct

verifier, but the code is associated with a code challenge that does not match this verifier.

PKCE ensures that an attacker cannot redeem a stolen authorization code at the token endpoint of the authorization server without knowledge of the code verifier.

PKCE is an OAuth extension, originally intended for securing public clients, but broader application to Authorization Code flow used by confidential clients is now recommended.

- Authorization codes must be one-time use and treated as invalid if already redeemed by the legitimate user. This leaves a limited period during which this flaw can be exploited. [23]
- Strictly bind authorization codes to a specific client. This may sound obvious, yet it deems exploitation impossible by injecting codes generated by another client, e.g. a malicious one. [23]

4.3.4 CSRF

Vulnerability

CSRF (Cross-Site Request Forgery) is a web security vulnerability that allows an attacker to induce users to perform actions that they do not intend to perform. It allows to partly circumvent the same origin policy, which is designed to prevent different websites from interfering with each other. [24]

In the context of OAuth 2.0 CSRF tricks a legitimate user into unknowingly accessing resources under the malicious user's control, which in specific implementations client implementations may lead to account takeover.

Consider a website that allows users to log in using either a classic, password-based mechanism or by linking their account to a social media profile using OAuth. Exploiting CSRF it is possible to hijack a victim user's account on the client application by binding it to their own social media account. [25][26]

Case-Study: Shopify login with Pinterest CSRF

Shopify's login with Pinterest feature was vulnerable to CSRF. [26] It was possible to bind a malicious Pinterest account to an existing Shopify account of a legitimate user by issuing an authorization code on Pinterest and tricking the authenticated Shopify user to visit a crafted site executing the following request on his/her behalf:

```
GET /auth/pinterest/callback?code=<code> HTTP/1.1
Host: pinterest-commerce.shopifyapps.com
...
Cookie: <shopify-session-cookie>
```

Note the lack of state parameter and/or PKCE code verifier on the above request.

Mitigations

- The "state" parameter should be utilized to link the authorization request with the callback request used to redeem an authorization code for an access token. This will ensure that the client is not tricked into completing any redirect callback unless it is linked to an authorization request initiated by the client.

The state value should be non-guessable, such as the hash of something tied to the user's session when the OAuth flow is initiated.

- Implementations PKCE extension also mitigates CSRFs against OAuth 2.0 as a side-effect.

4.3.5 Phishing and misleading user consent

Another important threat factor to be considered is the importance of user consent prompts and the risk of them misleading users and being abused in phishing campaigns attempting to trick the users into authorizing malicious OAuth applications. [27]

This phishing technique is hard to identify from an end user's perspective due to the fact that the user is never prompted to enter his credentials, nor is ever directed to a malicious site. For an attack of this type to succeed the victim user only has to authorize the malicious application, accepting the consent prompt hosted under the trusted domain of the authorization server.

Case-Study: Malicious "Google Docs" App Phishing Campaign

A phishing campaign leveraging a malicious Google application impersonating "Google Docs" and requesting access to user emails and

contacts was discovered in 2017. The phishing link was distributed by an email inviting the receiver to view a document and directed the victim to the consent page for the malicious application under the legitimate *accounts.google.com* domain. [28], [29], [30]

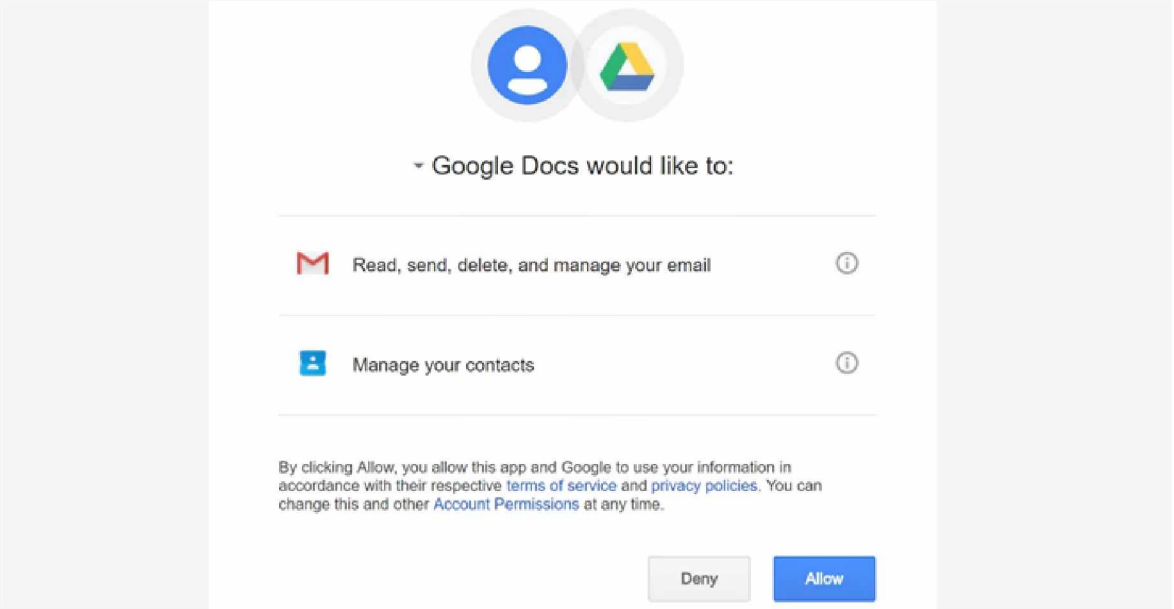

Figure 4.1: Google's consent prompt (at that time) for the malicious application. [15]

Countermeasures

- First and foremost a properly designed consent prompt providing the end user the required bits of information to review the legitimacy of the client.

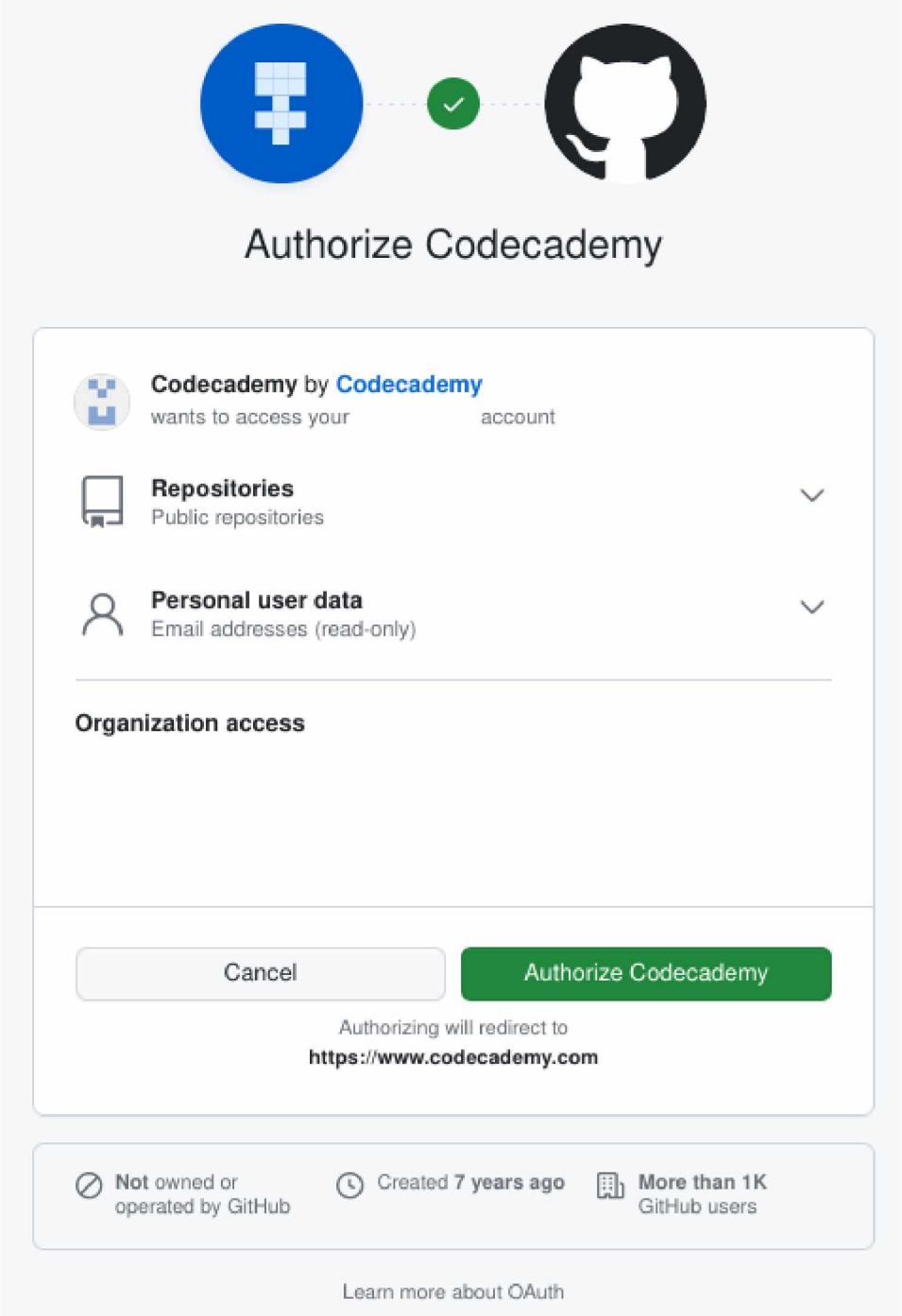

Figure 4.2: Github's consent prompt informing the user about the client's name, userbase, time of creation and association with Github as well as the requested scope and the redirect to follow authorization [31].

- Authorization servers have to provide end-users the capability to manage, audit and revoke access scope and authorization of client applications.
- Last but not least, considering leaked access and/or refresh token as a result of phishing attack or another attack vector is a realistic threat, token revocation features should be implemented to harden and support the operations of an Authorization server deployment. A mechanism for implementing JWT revocation will be discussed in the following chapter.

Chapter 5

Securing APIs with OAuth 2.0

In the previous chapters we have achieved a robust understanding of OAuth 2.0 framework as well as the relevant security implications. Considering emerging threats, their mitigation strategies and the risk they pose we are able to compile an auditing methodology for testing and verifying secure OAuth implementations.

Based on this methodology we are able to develop a proof of concept OAuth 2.0 system using containers, supporting different types of client applications and protecting a simple microservice application acting as a resource server.

We will further extend the system's baseline architecture to implement certain hardening features to ensure secure token management, transfer, verification and revocation while attempting to balance out security to performance trade-offs when securing APIs and microservices.

5.1 Development Environment

For development the above were deployed as Docker containers [32] within a Kubernetes [33] cluster exposed via the cluster's ingress controller [34]. OAuth Debugger [35] and OAuth Tools [36] served as client applications for testing and verification of the various OAuth flows.

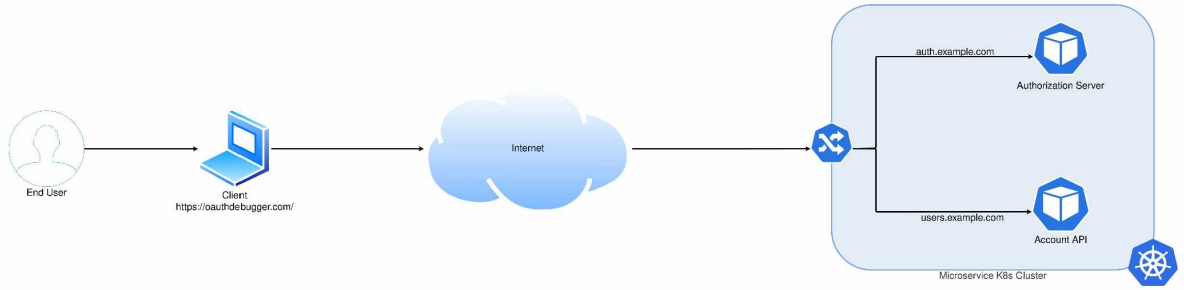

Figure 5.1: Development environment

5.2 High Level Architecture

The baseline system consists of an authorization server supporting confidential and public clients, implemented securely considering the topics discussed in chapters 3 and 4 which will be extended to support token revocation and introspection as well as an API microservice service to act as a protected resource server.

All access tokens issued are JWTs protected in transport by TLS.

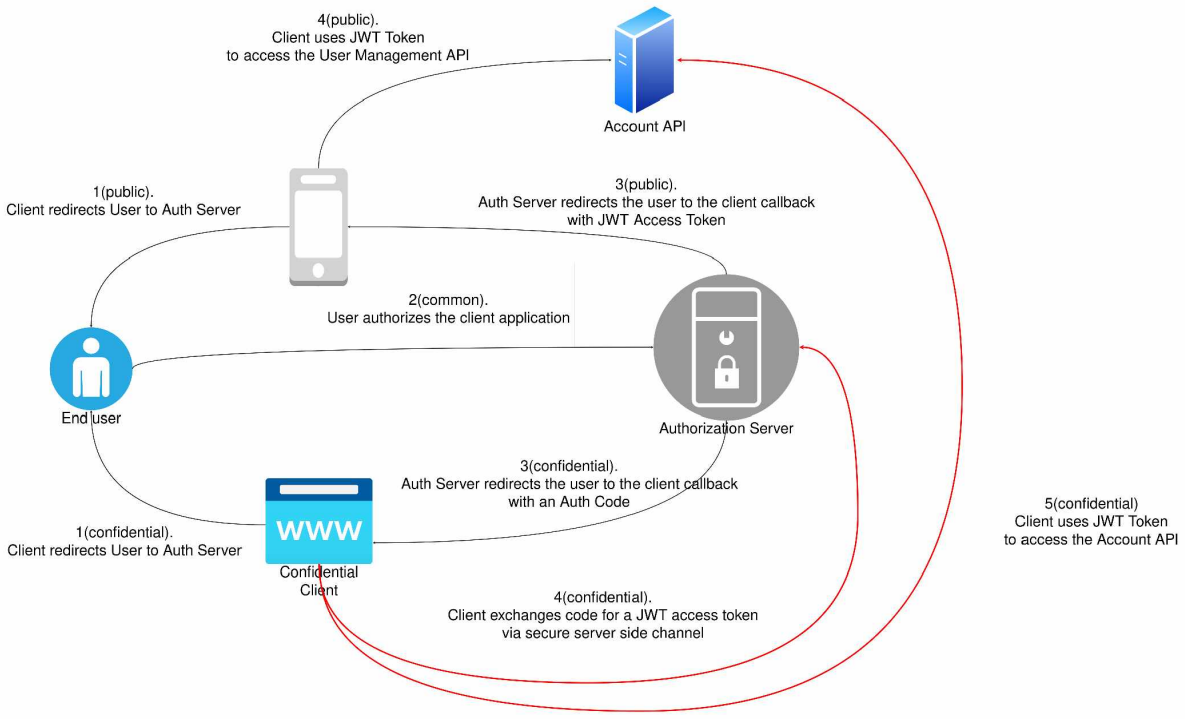

Figure 5.2: High level Architecture

5.2 JWT Revocation

Besides a robust authorization server implementation to properly secure OAuth protected resources it is crucial to implement access token revocation capabilities. The following scenarios can be referenced, in addition to the risks of improper token management and handling mentioned in 4.3.1, to demonstrate the necessity of invalidating an access token before its initially expected expiration time:

- a user revokes access to an authorized client application
- an application is deactivated by an authorization server admin
- a user is deactivated
- a user password is reset

Token revocation in implementations that utilize reference tokens stored in a database is straightforward. A token can be revoked by deleting its entry from the backend storage as such is the case of handling refresh tokens in parallel to JWT revocation discussed below.

Furthermore, a list of active access tokens can be retrieved at any time in order to audit the access granted on the user's behalf.

This is not the case for revoking self-encoded tokens such as JWTs which are designed to be portable, decoupled identity information that can be verified without interacting with the identity provider, who in the above implementation is the authorization server.

In this section the different ways for revoking JWTs [37], their pros and cons focusing on the "blacklist" approach to be implemented on top of the above infrastructure.

5.2.1 Short-lived access tokens

The most naive and simple approach would be issuing short-lived JWTs with a reduced validity period rather than implementing a revocation feature and accepting the potential risk of a significantly narrow abuse time frame.

Token revocation will not be instantaneous, but it could take up to the expiration of the last generated token. The main disadvantage of this approach is that all access tokens will expire periodically, which introduces a significant performance overhead even when a transparent token acquisition mechanism, like refresh tokens, is used by the backend to smoothen user experience.

This security to performance [38] tradeoff is considered acceptable and fair, even in the broader concept of session management, given the context of sensitive systems such as PCI-DSS compliant applications that demand session invalidation after 15 minutes of inactivity. [39]

5.2.2 Signing Secret Rotation

Another method for JWT revocation is to rotate the secret key used by the digital signing algorithm, invalidating every token signed with it. The obvious disadvantage of this approach is that it can not distinguish between individual clients, but revokes all access tokens at the same time.

Changing the signing key to implement common features like logout and session termination is extreme and does not fit most cases, yet in scenarios where there is a small and bounded number of active clients in the system, this overhead can be negligible considering the advantage of not being dependent on a centralized data storage and invalidation is instantaneous.

5.2.3 Token Blacklist

To implement JWT revocation blacklist to be able to tell one token apart from another one. To do so, an authorization server can utilize the uniquely valued `jti` (JWT's id) registered claim [40] used to identify a token. On an infrastructure level it is recommended using modern in-memory key-value storage like Redis [41] to implement the blacklist.

The main advantages of the blacklist revocation method is the fact that it can handle tokens individually and efficiently supports multi-client environments, allowing users to revoke access from different clients or devices on demand.

Token Revocation

To revoke a token an POST request is sent to the authorization server's revocation endpoint [42] as follows:

```
POST /revoke HTTP/1.1

Host: auth.example.com

Authorization: Bearer <token>

...

token=<jwt_token>&token_type=jwt
```

The authorization server validates the token and decodes its payload, extracts the `jti` claim to be used as a key for the blacklist entry and the `exp` claim to calculate how long the entry should be kept in the blacklist. By performing a lookup to the blacklist the authorization server can answer whether or not a non-expired JWT is revoked [43].

Blacklist Housekeeping

Scaling is the main limitation of the token blacklist as it can grow quite rapidly along the user-base. To mitigate this a maintenance mechanism (e.g. a cron job [44]) should execute ad-hoc on certain intervals, bounded by the JWTs lifetime, removing blacklist entries for expired tokens [45].

Implementation

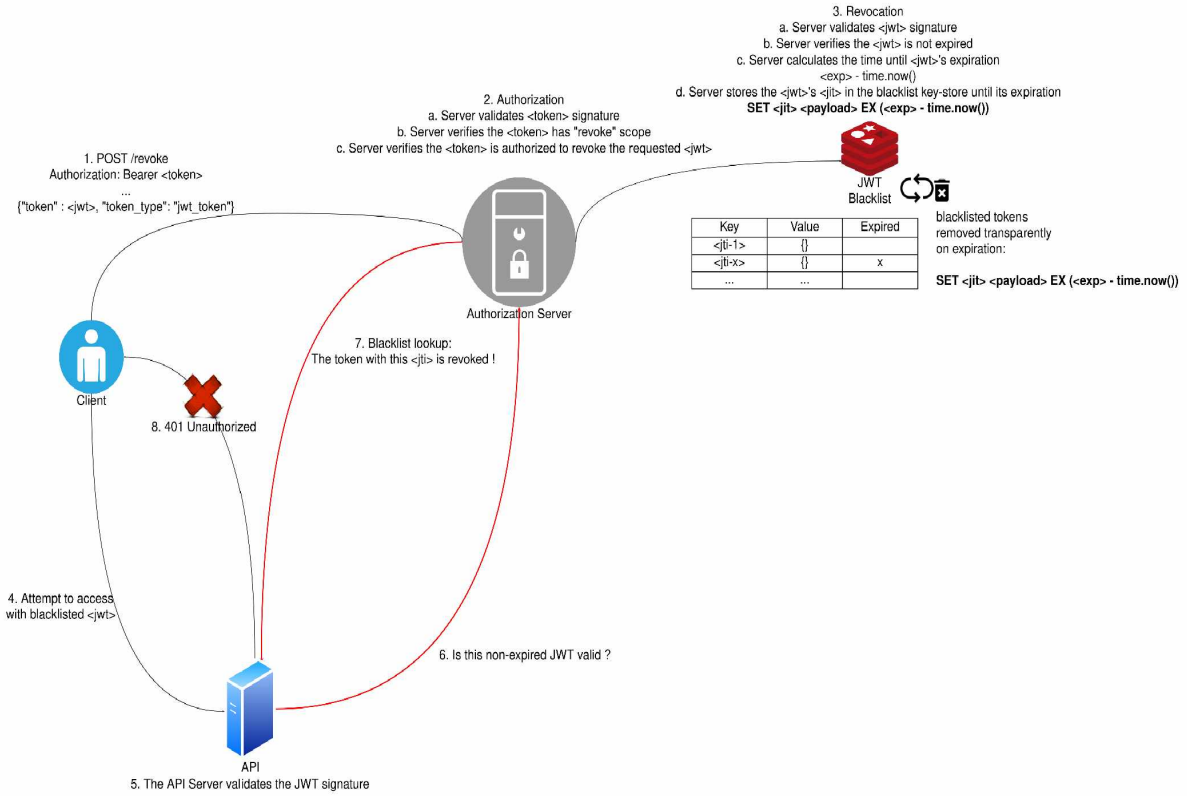

Figure 5.3: JWT revocation using REDIS as a blacklist

Since REDIS is used as the key-store solution to implement the blacklist, REDIS' the SET command with EX argument can be leveraged to implement the cleanup mechanism for deleting expired tokens. SET command sets a <key> to hold a certain <value> and the EX argument specifies the key's expiration time [46]. The key-value is removed upon expiration.

```
SET <jti> <payload> EX (<exp> - time.now())
```

The payload value can contain metadata like user identification information to support other features on top of the blacklist like *logout from all devices* or simply be empty JSON.

Security Considerations

Note that the blacklist stores very limited data, only non-sensitive information token metadata that can not be abused in case of compromise.

On the other hand the revocation endpoint is prone to exploitation and needs to be secured by defining a *revoke* scope that will be used to limit who can blacklist tokens only to clients holding this scope. This can prevent abuse scenarios of arbitrary revoking access tokens, aiming to disrupt users or exhaust blacklist resources.

Drawbacks

The key disadvantage of blacklist revocation is being contradictory with the distributed nature of JWTs, since it relies on lookups which introduce latency for validating tokens acting against performance benefits they offer.

In the following subchapter, the proof of concept implementation, now supporting JWT revocation will be extended further in an attempt to minimize this performance overhead and get the best of both worlds.

5.2.4 Other JWT revocation methods

There are a couple less prevalent approaches based on distributing revocation events [47] broadcasted from the authorization server to all resource servers. These revocation methods come with significant drawbacks, the dominant ones being not supporting multiple client applications [45] and demanding complex logic to be implemented on the resource server side and as they are not standardized will not be analyzed here.

5.3 JWT Lifetime and Validation

Given revocation capabilities introduces certain issues when validating a JWT. In order to distinguish between the 3 different states a JWT may be, an API server can perform the following actions before accepting it.

Figure 5.4: JWT Token Lifetime

5.3.1 Local Validation

Fast or local validation is the customary, well established validation method for JWTs which holds all the performance benefits discussed in earlier chapters. It can be performed locally by the API server accepting JWT access tokens as follows:

1. `Inspect JWT`
2. `Check the expiration timestamp`
3. `Validate the cryptographic signature`

The public key can be fetched from the issuing authorization server's metadata endpoint [48] and cached locally.

5.3.2 Remote Validation

Local validation is not sufficient for identifying a revoked JWT. To do so the API server has to validate against the revocation blacklist maintained by the authorization server to strongly validate a token. This can be done using the authorization server's token introspection endpoint [49], meant to determine the active state meta-information of an issued token. However this requires performing a network request, hence introduces latency.

Request:

```
POST /introspect HTTP/1.1
Host: auth.example.com
Authorization: Bearer <token>
…
token=<jwt_token>
```

Response:

```
HTTP/1.1 200 OK
…
{"active": true,
"jti": "<jti>",
…
"scope": "read-profile write-profile",
"exp": 1419356238}
```

5.4 Securing API backends

In the context of an API or a microservice application the various operations hold different levels of criticality. Assessing the risk of each operation can be used to define whether or not remotely validating an access token is a strong requirement and justifies the performance impact.

For example fetching an avatar from the Profile API using a revoked token does not pose the same impact as changing a user password or accessing the Payments API and executing a transaction. The latter should strongly validate the access token, while the first operation can be processed with only performing local validation.

5.4.1 API Gateway Pattern

To demonstrate this principle we are going to extend the proof of concept implementation with an API Gateway. An API Gateway is a microservice architecture pattern where an infrastructure component or software middleware acts as an entry point for client applications to access a collection of backend services or APIs [50].

The main purpose of the API Gateway component serves is managing and implementing centrally using a unified interface key features such as:

- Request routing and load balancing
- Authentication and authorization
- Security features like API schema enforcement
- Rate limiting and throttling

- Logging, monitoring and tracing

Implementation

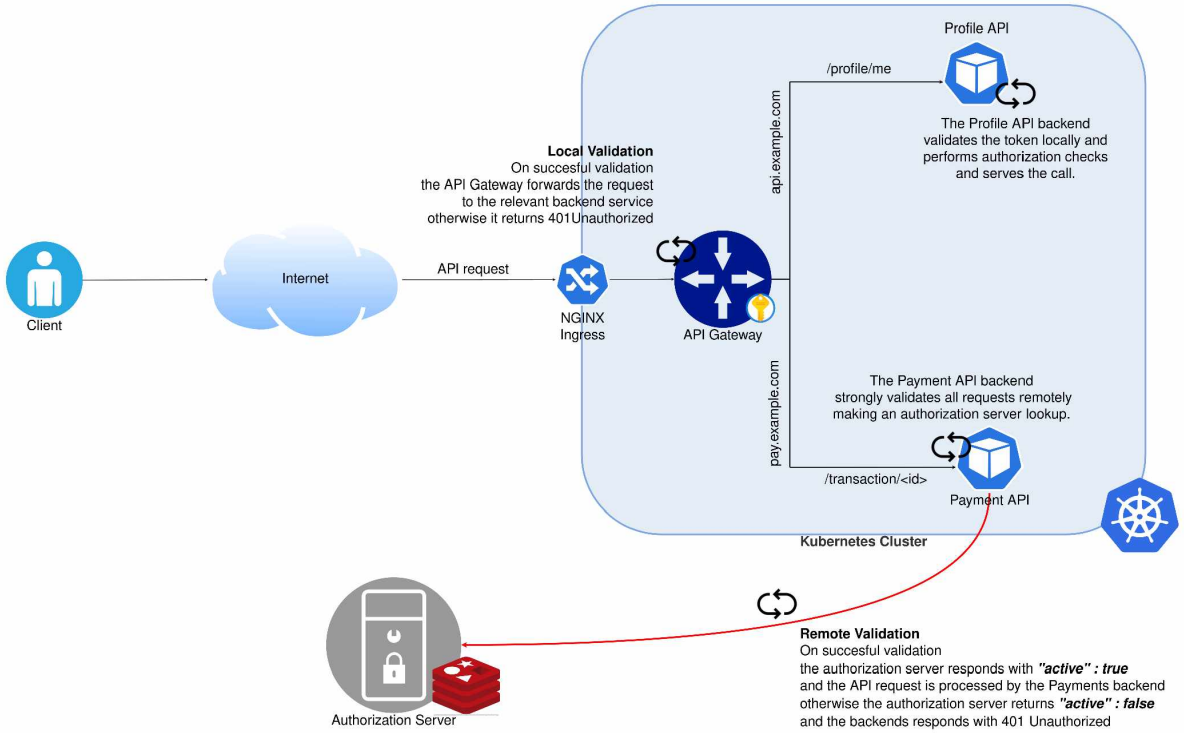

Figure 5.5: Securing backend services using an API Gateway

We are going to leverage the API gateway to balance out the cost in performance introduced by strong token validation by reducing the number of requests processed within the internal network.

The API Gateway is going to act as a boundary between the microservice network and the clients, performing local validation on all access tokens contained in authenticated API requests before they reach the respective backend service. Requests with invalid, tampered or expired access tokens will be dropped on the network's edge, omitting unnecessary processing and resource commitment by the internal microservice infrastructure.

This leaves it to the different backend service APIs to choose whether or not to perform strong access token validation based on the risk posed by the requested action.

Security Considerations

It is important to note that this is not a security feature, rather a performance optimization, nor does it act as an alternative to authorization checks that have to be performed on the backend services.

Furthermore, it is recommended to perform local validation on the API service before blindly accepting a JWT, unless it is possible to validate that the received request source is the API gateway.

This can be done by enforcing network policies or extending the API gateway to act as an authwall boundary, utilizing HTTP headers to pass token metadata to backend services and service mess patterns for TLS and service identities [51].

Chapter 6

Conclusions and Future Work

6.1 Summary

We presented an overview of the OAuth 2.0 framework along with the necessary background information for understanding the problem it aims to solve and its limitations. Diving deeper into the framework's internals we used a simple threat model and specific implementation case studies to identify the emerging threats within OAuth 2.0 and their impact, along with the risk they pose and their potential mitigation strategies.

Based on the above we compiled an audit methodology which assisted us into implementing a proof-of-concept OAuth based system to protect a simple microservice application API utilizing JWT tokens. Finally the proof of concept architecture was extended with infrastructure components to further harden the systems token management capabilities. Token revocation is supported and leveraged by backend services when required, dealing with the security and performance implications that emerge from the use of JWTs.

6.2 Future Work

In the future the review methodology of OAuth implementations should be improved to consider OAuth 2.1 specification and security considerations [52]. The latest draft versions available at the time [53],[54] of writing were considered but should be revisited once they are standardized. A more comprehensive audit of OAuth should also consider implementations that leverage authorization server chains, attack vectors against JWTs [55] and mobile clients as well as underestimated clickjacking [56] and flow downgrade attacks.

As of infrastructure we can further utilize the API Gateway to explore microservice architectural patterns or implement other security and performance features. For example implementing a service mesh [57] within the microservice network allows verifying the legitimacy of a request's source with mTLS, negating the need of re-validating the JWT locally on the API's backend [51].

References

[1] https://en.wikipedia.org/wiki/Identity_management

[2] https://en.wikipedia.org/wiki/Session_(computer_science)#Session_management

[3] https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies

[4] https://en.wikipedia.org/wiki/Access_token

[5] https://www.okta.com/identity-101/what-is-token-based-authentication/

[6] https://datatracker.ietf.org/doc/html/rfc7519

[7] https://jwt.io/

[8] https://fusionauth.io/articles/tokens/pros-and-cons-of-jwts

[9] https://www.rfc-editor.org/rfc/rfc6749#section-1.3.4

[10] https://www.rfc-editor.org/rfc/rfc8628

[11] https://www.ietf.org/archive/id/draft-ietf-oauth-v2-1-08.html#name-differences-from-oauth-20

[12] https://www.rfc-editor.org/rfc/rfc6749#section-1.3.3

[13] https://www.rfc-editor.org/rfc/rfc6749#section-1.5

[14] https://datatracker.ietf.org/doc/html/rfc6749#section-1.3.1

[15] https://datatracker.ietf.org/doc/html/rfc6749#section-1.3.2

[16] https://datatracker.ietf.org/doc/html/draft-ietf-oauth-security-topics-22#name-implicit-grant

[17] https://oauth.net/2/pkce/

[18] https://datatracker.ietf.org/doc/html/rfc6819

[19] https://datatracker.ietf.org/doc/html/draft-ietf-oauth-security-topics#name-insufficient-redirect-uri-v

[20] https://hackerone.com/reports/2575

[21] https://datatracker.ietf.org/doc/html/draft-ietf-oauth-security-topics#name-countermeasures

[22] https://datatracker.ietf.org/doc/html/draft-ietf-oauth-security-topics-23#name-authorization-code-injectio

[23] https://datatracker.ietf.org/doc/html/draft-ietf-oauth-security-topics-23#name-discussion

[24] https://portswigger.net/web-security/csrf

[25] https://portswigger.net/web-security/oauth#flawed-csrf-protection

[26] https://hackerone.com/reports/111218

[27] https://www.youtube.com/watch?v=espX8qKjywI

[28] https://twitter.com/googledocs/status/859878989250215937

[29] https://auth0.com/blog/all-you-need-to-know-about-the-google-docs-phishing-attack/

[30] https://thehackernews.com/2017/05/google-docs-phishing-email.html

[31] https://docs.github.com/en/apps/oauth-apps/using-oauth-apps/authorizing-oauth-apps

[32] https://www.docker.com/

[33] https://kubernetes.io/

[34] https://kubernetes.io/docs/concepts/services-networking/ingress/

[35] https://oauthdebugger.com/

[36] https://oauth.tools/

[37] https://journals.sagepub.com/doi/full/10.1177/1550147718801535

[38] https://auth0.com/blog/balance-user-experience-and-security-to-retain-customers/

[39] https://otm.finance.harvard.edu/files/otm/files/pci_security_standards.pdf - Section 8.5.15

[40] https://www.rfc-editor.org/rfc/rfc7519#section-4.1.7

[41] https://redis.io/

[42] https://auth0.com/blog/denylist-json-web-token-api-keys/

[43] https://datatracker.ietf.org/doc/html/rfc7009#section-2

[44] https://en.wikipedia.org/wiki/Cron

[45] http://waiting-for-dev.github.io/blog/2017/01/24/jwt_revocation_strategies

[46] https://redis.io/commands/set/

[47] https://fusionauth.io/articles/tokens/revoking-jwts

[48] https://www.rfc-editor.org/rfc/rfc8414

[49] https://datatracker.ietf.org/doc/html/rfc7662

[50] https://www.nginx.com/learn/api-gateway/

[51] ttps://fusionauth.io/articles/tokens/tokens-microservices-boundaries

[52] https://datatracker.ietf.org/doc/html/draft-ietf-oauth-v2-1-08

[53] https://fusionauth.io/articles/oauth/differences-between-oauth-2-oauth-2-1

[54] https://datatracker.ietf.org/doc/html/draft-ietf-oauth-security-topics-23

[55] https://portswigger.net/web-security/jwt

[56] https://owasp.org/www-community/attacks/Clickjacking

[57] https://en.wikipedia.org/wiki/Service_mesh

[58] https://istio.io/latest/about/service-mesh/

[59] draw.io