Aalto University
School of Science

Master’s Programme in Computer, Communication and Information Sciences

Securing web clients utilizing OAuth 2.0

Jonas Bäck

Master’s Thesis
2024

Aalto University
School of Science

Author Jonas Bäck

Title Securing web clients utilizing OAuth 2.0

Degree programme Computer, Communication and Information Sciences

Major Security and Cloud computing

Supervisor Prof. Antti Ylä-Jääski

Advisor M.Sc. Karri Lehtiranta

Collaborative partner twoday

Date 3.6.2024 **Number of pages** 57 **Language** English

Abstract

The OAuth 2.0 specification is widely used to authorize applications to use resources belonging to third parties and to enable single sign-on across multiple applications. While the technology is widely used, the lax nature of the specification leaves plenty of different options for developers looking to implement the specification, enabling different OAuth systems to have very different security characteristics. As third-party OAuth authorization servers are often utilized by developers that might have limited knowledge of the technology and might thus be unaware of the security implications of their choices, this thesis aims to present known vulnerabilities in OAuth systems, and demonstrates simple steps that should be taken to securely implement the specification.

This thesis presents a number of vulnerabilities of varying severity that are common in real-world systems utilizing OAuth. These vulnerabilities are often caused by incorrect implementation of OAuth clients and authorization servers, but the lax nature of the specification allowing the creation of insecure applications if only the bare minimum of the specification is implemented is an equally common source for vulnerabilities. Through a practical client implementation utilizing third-party authorization servers, this thesis shows that a number of simple steps can be taken to mitigate many of the described vulnerabilities, often at a minimal development and performance cost and without a significant increase in complexity. It is however not always obvious if the most secure option is the optimal choice for a specific system, such as when deciding if and how to persist user sessions, as a more secure option might lead to a deterioration in user experience. To help developers adapt the OAuth specification to their specific application according to their security requirements, this thesis presents the impact of different design choices and shows why taking extra precautions is often worth the effort.

Keywords OAuth 2.0, single sign-on, web development, security, authentication, authorization

Aalto-universitetet
Högskolan för
teknikvetenskaper

Författare Jonas Bäck

Titel Säkrande av webbklienter som utnyttjar OAuth 2.0

Utbildningsprogram Computer, Communication and Information Sciences

Huvudämne Security and Cloud computing

Övervakare Prof. Antti Ylä-Jääski

Handledare D.I. Karri Lehtiranta

Samarbetspartner twoday

Datum 3.6.2024 **Sidantal** 57 **Språk** Engelska

Sammandrag

OAuth 2.0-specifikationen används ofta för att auktorisera applikationer att använda resurser som tillhör tredje parter och för att möjliggöra att användare autentiseras med samma konto i flera applikationer. Även om teknologin används i stor utsträckning ställer specifikationen få krav på utvecklare, vilket möjliggör många olika alternativ då teknologin implementeras. Denna brist på strikta krav gör det möjligt för varierande applikationer att utnyttja teknologin, men det leder också till att olika OAuth-system kan ha väldigt olika säkerhetsegenskaper. OAuth auktoriseringsservrar som upprätthålls av tredje parter utnyttjas ofta av utvecklare för autentisering, vilket leder till att utvecklare slipper implementera komplicerade inloggningssystem för att autentisera sina användare. Då utvecklare kan utnyttja OAuth servrar och implementera OAuth klienter utan djupare kunskap om teknologin är det lätt hänt att beslut om hur systemet ska implementeras tas utan att ta säkerhetskonsekvenserna i beaktande. Målet med denna avhandling är att presentera kända sårbarheter i OAuth-system och beskriva enkla steg som bör tas för att implementera specifikationen på ett säkert sätt.

I denna avhandling presenteras ett antal sårbarheter av varierande allvarlighetsgrad som är vanliga i system som använder OAuth. Dessa sårbarheter orsakas ofta av felaktig implementering av OAuth-klienter och auktoriseringsservrar, men bristen på strikta krav i specifikationen som gör det möjligt att skapa osäkra applikationer utan att göra explicita fel är en lika vanlig källa till sårbarheter. Genom en praktisk implementation av en OAuth-klient som utnyttjar tredje parters auktoriseringsservrar visar denna avhandling att ett antal enkla steg kan tas för att minska många av de beskrivna sårbarheterna, ofta till en minimal kostnad och utan en betydande ökning av komplexitet. Det är dock inte alltid uppenbart om det säkraste alternativet är det optimala valet för ett visst system, exempelvis då man ska besluta om och hur användarsessioner ska bevaras, eftersom ett säkrare alternativ kan leda till en försämrad användarupplevelse. För att hjälpa utvecklare att anpassa OAuth-specifikationen till sin specifika applikation i enlighet med sina säkerhetskrav presenterar avhandlingen effekterna av olika beslut då OAuth-system implementeras och visar varför det ofta är värt besväret att vidta extra åtgärder för att göra system tillräckligt säkra.

Nyckelord OAuth 2.0, webbutveckling, säkerhet, autentisering, auktorisering

Contents

Abstract 3

Abstract (in Swedish) 4

Contents 5

Abbreviations 7

1 Introduction 8

- 1.1 Problem statement 8
- 1.2 Structure of the thesis 9

2 Background 10

- 2.1 Securing network traffic 10
  - 2.1.1 Public key encryption 10
  - 2.1.2 Transport Layer Security 11
- 2.2 Structure of web applications 13
  - 2.2.1 Static web pages 13
  - 2.2.2 Rich internet applications 14
  - 2.2.3 Single-page applications 15
- 2.3 Client-side storage 16
  - 2.3.1 Cookies 17
  - 2.3.2 Web storage API 17
- 2.4 Authenticating HTTP requests 18
- 2.5 Cross-site request forgery 19

3 The OAuth 2.0 specification 21

- 3.1 Roles 21
- 3.2 Identifiers 22
- 3.3 Authorization flows 23
  - 3.3.1 Authorization code grant 23
  - 3.3.2 Implicit grant 25

4 Vulnerabilities in OAuth systems 27

- 4.1 Vulnerabilities in authorization servers 27
  - 4.1.1 HTTP 307 redirect 27
  - 4.1.2 Insecure token handling 28
  - 4.1.3 Redirect URI attack 29
- 4.2 Vulnerabilities in OAuth clients 29
  - 4.2.1 Invalid state handling 29
  - 4.2.2 IdP mix-up attack 31
  - 4.2.3 Vulnerabilities in token storage 31

5 Implementing an OAuth client **34**
5.1 Application structure 34
5.1.1 Obtaining an access token 34
5.1.2 Inspecting existing access tokens 43
5.1.3 Refreshing access tokens 43
5.2 Token storage 43
5.3 Proof Key for Code Exchange 45
5.4 Explicit user intention tracking 46

6 Discussion **47**
6.1 Client implementation options 47
6.1.1 OAuth authorization flows 47
6.1.2 Persisting OAuth tokens 48
6.2 Observations from utilizing third-party authorization servers 49
6.2.1 Quality of documentation 49
6.2.2 Authorization server security 50

7 Conclusions **52**

Abbreviations

HTTP	Hypertext transfer protocol
HTTPS	Hypertext transfer protocol secure
TLS	Transport layer security
OIDC	OpenID connect
SSO	Single sign-on
HTML	HyperText markup language
CSS	Cascading style sheets
SEO	Search engine optimization
RIA	Rich internet application
SPA	Single-page application
JSON	JavaScript object notation
Ajax	Asynchronous JavaScript and XML
API	Application programming interface
IdP	Identity provider
MFA	Multi-factor authentication
URI	Uniform resource identifier
XSS	Cross-site scripting
PKCE	Proof key for code exchange
CSRF	Cross site request forgery

1 Introduction

In recent years, single sign-on (SSO) systems have become common, allowing users to authenticate using one centralized account. SSO systems can help secure user authentication by mitigating password fatigue and password reuse [1], in addition to creating a more convenient user experience by removing the need for the user to authenticate on each web application separately. Additionally, SSO systems remove the need for each web application to implement their own authentication system, allowing a smaller number of centralized actors with the required resources to implement the systems securely.

OAuth 2.0 is an open standard that is commonly used by SSO systems that allow users to share information about their account with third parties, allowing the third party to use the SSO system to authenticate their users [2]. The OAuth specification defines how authorization servers function and how they should be interacted with. The specification does not however define how the client interacting with the authorization server should be implemented, leaving room for potential developer mistakes and unintended vulnerabilities. The popularity of OAuth makes it a tempting attack vector, further increasing the chance that any vulnerabilities in the client implementation will be exploited.

1.1 Problem statement

The OAuth 2.0 specification is widely used today. The specification is however not very strict, leaving plenty of room for developers to implement the system as they like. While this does enable a wide variety of systems to utilize OAuth, it means that two different systems implementing the same specification might not be as secure. As the specification is quite well studied, a number of common vulnerabilities are known, with well defined defenses. Some of these defenses are described as recommendations in the OAuth specification, while other solutions might only be described in third-party sources.

As SSO-systems are typically used by developers to lower the required amount of logic needed to implement authentication, it is not only essential that it is easy to implement an OAuth client, but that it is easy to implement a secure client. There are a number of different options that are available when implementing an OAuth client, and it is important to consider the security implications of these decisions. While the specification might allow certain implementation options, understanding the negative effect on the system's security should help developers consider other options, leading to a more secure system. Multiple widely used open-source OAuth clients exist, but their implementations can vary significantly and the motivation behind their choices is not always well defined. Additionally, the OAuth clients are often implemented by and for a specific identity provider, forcing developers to implement their own client or combining multiple clients if they want to support multiple providers. Different client platforms also come with specific security considerations; this thesis will however focus on JavaScript-based web clients.

This thesis aims to achieve the following objectives:

1. **Identify common vulnerabilities in OAuth clients**
   To be able to securely implement OAuth-based authentication, it is essential to understand common vulnerabilities, both in OAuth clients and authorization servers.

2. **Demonstrate a secure client implementation**
   The second objective is to implement a secure OAuth web-client using JavaScript and to demonstrate how the previously shown vulnerabilities are mitigated.

1.2 Structure of the thesis

Chapter 2 offers an overview of relevant technologies when inspecting the security of OAuth web clients, providing information on how web traffic is secured and how web applications are typically structured. Chapter 3 gives an overview of the OAuth 2.0 specification and describes how it can be used to authenticate users. Chapter 4 investigates common vulnerabilities in OAuth systems. Chapter 5 presents a JavaScript-based OAuth client which attempts to mitigate the vulnerabilities found in chapter 4. Chapter 6 discusses the implementation of chapter 5 and how it takes the vulnerabilities of chapter 4 into account. The chapter also lists observations from utilizing a third-party authorization server, focusing on the quality of documentation as well as on how the authorization servers have been implemented. Chapter 7 summarizes the key results of the study.

2 Background

This section gives an overview of the areas that are required to evaluate the security of an OAuth system. Since the OAuth specification deals with transmitting sensitive data over the internet, the first section describes how network communication can be secured, with details on public key encryption and Transport Layer Security. The second section describes the structure of web applications, considering the different implementation options that are available today as well as their security considerations. To be able to evaluate different methods for storing OAuth user identifiers, the third section details the different options that are available for storing data in web clients. The fourth section describes different methods that can be used to authenticate HTTP communication. The fifth section describes cross-site request forgery attacks, which is a common attack type that is especially relevant in the context of OAuth.

2.1 Securing network traffic

The complex and distributed nature of the internet causes it to be inherently insecure. A network request may travel through switches located in a number of countries and operated by various actors, and the route might even change between requests as the network topology and load changes. As such, we have to assume that some link along the chain is listening to the traffic we send. In the context of web traffic, this lack of trust causes issues, as users want to be able to send potentially sensitive information with the knowledge that no external actor will be able to read or modify the data.

2.1.1 Public key encryption

To mitigate the risk of malicious parties being able to listen in on network communication, public-key encryption can be used. Public key cryptography relies on a public-private key pair, where the public key is used to encrypt data and the private key is used to decrypt data. A network actor can generate the key pair and send the public key to the other endpoint. Since the public key can only be used to encrypt data, an attacker will not gain any valuable information if they are able to intercept the public key. With the public key, the other endpoint can encrypt the data before sending it over a network. Since the data can only be decrypted by the corresponding private key, a network attacker which successfully intercepts the communication is unable to read the original data [3].

Diffie-Hellman key exchange is one method to exchange keys for public key cryptography [4] [5].

1. The Diffie-Hellman key exchange begins by two parties, Alice and Bob, agreeing on two publicly shared values, p, and g, where p is a prime and g is a primitive root modulo p.

2. Alice chooses a secret integer a and sends $A = g^a \mod p$ to Bob.

3. Bob chooses a secret value b and sends $B = g^b \mod p$ to Alice.

4. Alice computes $s = B^a \mod p$
5. Bob computes $s = A^b \mod p$

Alice and Bob now share the secret s, since

$$A^b \mod p = g^{ab} \mod p = g^{ba} \mod p = B^a \mod p$$

The secret share is considered secure, as finding a and b given $g^{ab} \mod p = g^{ba} \mod p$ takes a long time to compute with our current algorithms.

If Diffie-Hellman key exchange is used, an attacker can store large amounts of network traffic, with the hope of the secret key leaking in the future, thus allowing the attacker to decrypt all previous communication. Additionally, in traditional public-key cryptography, the keys between two parties can stay constant for long amounts of time, allowing an attacker to decrypt all previous traffic at once if the secret is leaked in the future. To protect against such attacks, short lived keys, commonly referred to as ephemeral keys, can be used, such as in the SAKE protocol [6]. The SAKE protocol solves the issue of a future key leak being able to decrypt previous communication by continuously updating the secret key, making it impossible to derive an older secret key from a newer one. Ensuring that an encryption scheme is resistant to keys being leaked at a later point is known as perfect forward secrecy.

2.1.2 Transport Layer Security

Today, the Transport Layer Security (TLS) protocol is used to prevent eavesdropping, tampering and forging messages in client-server communication. The latest version of the specification is TLS 1.3, which among other things removes insecure ciphers, allows for fewer round trips in the handshake and forces clients and servers to use ephemeral key exchange methods, ensuring perfect forward secrecy [7].

TLS provides a secure channel which provides authentication, confidentiality and integrity, even if an attacker has complete control of the network. The server is always authenticated, while the client can be optionally authenticated. The confidentiality of the transmitted data is ensured by encryption, only allowing the server and client to decrypt the contents. TLS 1.3 additionally ensures forward secrecy, effectively preventing attacks that store large amounts of encrypted data with the hope of decrypting it at a later point when keys are compromised. The lengths of the data and the identities of the client and server are however not hidden. Integrity is ensured by detecting any external modifications to the sent data.

The full TLS 1.3 handshake, described in figure 1, functions as follows [8]:

1. The *ClientHello* message is used for negotiating parameters and exchanging keys. The configurable parameters are TLS versions, groups, cipher suites and certificate authorities. The key exchange is initiated by sending the client's Diffie-Hellman key share. The client can request a server certificate, further ensuring that no impersonation attack can take place.

2. *ServerHello*: The server selects a cipher and TLS version based on the lists sent by the client and confirms that the handshake is finished. The server can suggest new parameters if it does not support any of the parameters sent by the client. The server also sends its own Diffie-Hellman key share and a server certificate, if one was requested. If client authentication is used, the server will request a client certificate. If a pre-shared key is used, the server can send application data in this message, allowing for data transfer without any additional round trips.

3. A secure channel is established, allowing the client to send application data encrypted with the previously shared key. The client confirms that the handshake has been completed and sends a certificate if one was requested.

4. The server can now use the secure channel to send encrypted application data.

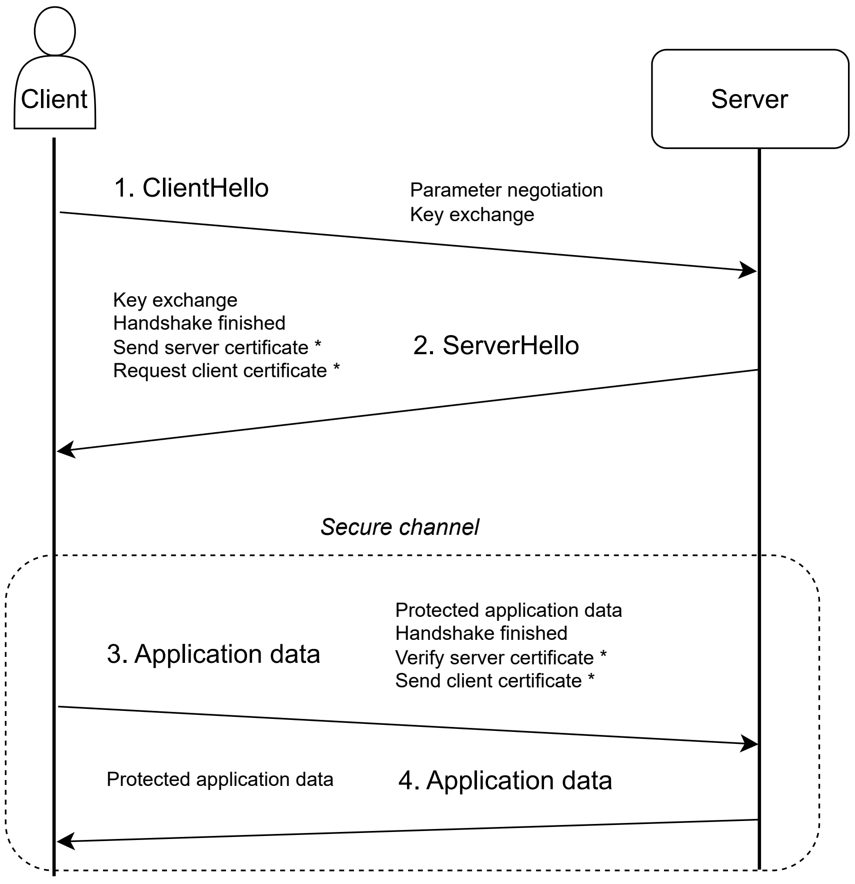


Figure 1: Full TLS 1.3 handshake. Optional content marked with *.

2.2 Structure of web applications

Web pages can be characterized by clients fetching web pages from servers and displaying the pages in web browsers. The basic building blocks for a web page are HyperText Markup Language (HTML) elements, which describe the structure and content of the page. The layout and style of web pages can be modified by using Cascading Style Sheets (CSS). The addition of JavaScript makes it possible for the browser to execute arbitrary code, greatly improving the options for interactive pages. Since its inception in the 80's, the web has evolved and matured. From static, read-only pages utilizing HTML to pages allowing users to write and participate in the web during the turn of the millennium, made possible in part by the adoption of JavaScript. The evolution continued to allow users to execute programs that we commonly refer to as web applications today [9].

2.2.1 Static web pages

A traditional, static web page is stored in its entirety on the server and is served to users on request. Such pages typically require only HTML and CSS to function. As the client side barely includes any logic, the focus is on the server, which stores and serves data and handles all business logic. Navigation and interaction with the page is done using clickable links that direct the user to other pages. Figure 2 describes a user visiting a web page *example.com*, clicking a link to visit *example.com/apply* and finally submitting a form to *example.com/apply*. Keeping the content strictly separated in pages and serving the complete page as-is to clients provides a number of benefits [10]:

- The amount of data transferred when visiting a site is low, as only the content that is immediately shown to the user has to be sent. This speeds up load times for the end user, reduces data usage and typically leads to a lower delay before content is visible in the browser.
- Static websites are typically easier to traverse programmatically than dynamic sites. This makes it easier for web crawlers to index the site, such as those used by search engines, making search engine optimization (SEO) trivial. Additionally, static sites are typically more accessible as they can be parsed more easily by screen readers and similar software [11].
- As the use of JavaScript allows websites to execute near arbitrary code in browsers, users can choose to disable JavaScript altogether. This will cause most dynamic sites to stop working, while static websites can function without using JavaScript. This removes the option for potential attackers to execute harmful code on the user's machine and often limits the tracking capabilities of websites.

There are, however, reasons why strictly static sites are rare today, compared to the early days of the web when they were the standard [12]:

- Static websites are slow and clunky to use, as any interaction with the site requires the client to fetch a new page, or reload the current page from the server.
- Static websites only support client-initiated requests, making it impossible for the server to implement event-driven patterns, such as sending notifications to the client.
- Continually updating pages is not possible, as doing so would require server-initiated communication. The problem could be solved by the client periodically requesting the latest data, but this is not possible in strictly static sites as it requires JavaScript to implement.

2.2.2 Rich internet applications

To allow for client-side scripting and alleviate some of the issues with strictly static sites, JavaScript was introduced in the mid 90's. While this allowed developers to create dynamic user interfaces, web apps as we know them today did not become popular until the introduction of third-party tools, such as Adobe Flash and Java in web

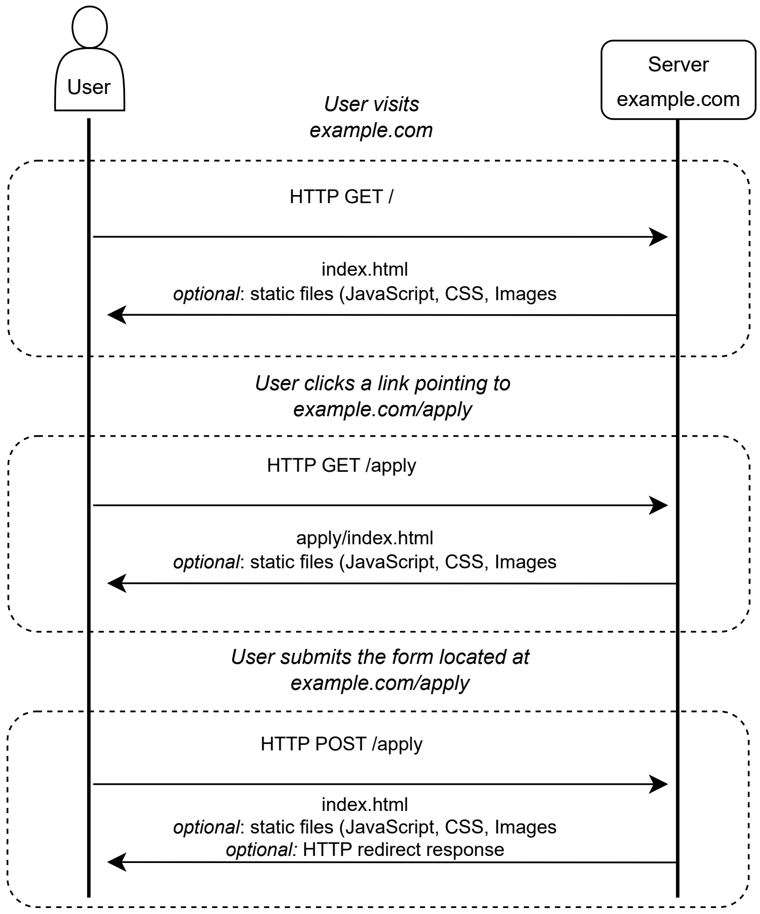

User
Server
example.com
User visits
example.com
HTTP GET /
index.html
optional: static files (JavaScript, CSS, Images
User clicks a link pointing to
example.com/apply
HTTP GET /apply
apply/index.html
optional: static files (JavaScript, CSS, Images
User submits the form located at
example.com/apply
HTTP POST /apply
index.html
optional: static files (JavaScript, CSS, Images
optional: HTTP redirect response

Figure 2: Network traffic for a user interacting with a static web site.

apps. These technologies allowed for the development of rich internet applications (RIAs) that delegated some logic and data storage to the client, while using separate HTML pages for different pages [13]. While these tools allowed for the creation of interactive web apps, end users had to install the tools separately if they wanted to utilize them. In addition to being a hassle for end users, these plugins introduced additional security vulnerabilities. At the time of writing, 1084 common vulnerabilities and exposures (CVEs) have been reported for Adobe Flash [14].

Some way to create interactive web apps using only HTML, CSS and JavaScript was needed. The adoption of Asynchronous JavaScript and XML (Ajax) around 2005 saw web development move towards JavaScript-based applications that are common today. The asynchronous nature of Ajax allowed for completing network requests in the background without blocking user input and updating relevant parts of the interface when the request completed. JavaScript-based abstractions built on top of Ajax made development easier, such as jQuery [15], released in 2006, and the fetch() function [16], released in 2015.

2.2.3 Single-page applications

Logic can further be delegated to the client, allowing for single-page applications (SPAs), that contain all the web pages for an application in one HTML page [17]. While the name might imply that SPAs only have one page, they can have any number of pages visible from the end users point of view, with the term *single page* referring to the number of HTML pages. In contrast to static web applications and RIAs, SPAs use client side logic instead of full HTTP requests to handle navigation and user input. This often leads to improved responsiveness for users and makes dealing with client-side data and state easier, at the cost of a longer initial load time due to the larger size of the page. The backend typically consists of an application programming interface (API), which is designed for machine communication with responses that might not be easily human-readable. The backend and frontend often utilize JavaScript Object Notation (JSON) data to communicate.

Figure 3 shows the network traffic between a user and a web server for a simple SPA. When comparing the traffic with the static page traffic in figure 2, the static page always returns a complete HTML page and redirects the user accordingly, while the SPA only fetches one HTML page on the initial page load. The requests for the user navigating to */apply* and the user submitting a form can be identical for both the static page and the SPA. The response from the backend is however different, with the static application returning a complete page and the SPA returning data in any suitable format, typically JSON.

While the network traffic of the SPA and static web page might seem very similar at first, there are many practical differences [17]. As SPAs implement navigation in the client, user navigation could be achieved without any network traffic, completely relying on the content of the first page load. In practice some network traffic is often useful when navigating a SPA. Dynamic information can be fetched on navigation, periodically by the client or on-demand by the server, as data fetched on initial page load could easily become stale. Pre-rendered HTML and static content such as images can also be fetched on navigation, lowering the size of the initial HTML page. With SPAs, developers are given greater freedom in implementation, having the option to develop web applications very similarly to native applications, while not requiring end users to install anything on their machines. While moving much of the logic to the client can make applications more responsive, moving logic from an opaque server to a transparent client application requires developers to pay attention to which parts they could move to the client. In general, developers should assume that all data and application logic that is located in the client can be freely accessed and modified by potential attackers.

2.3 Client-side storage

In web applications that execute code on the client, storing data in the client browser is often required. The use of JavaScript allows developers to store data in variables, making it possible to keep track of application state and to create more complex programs. It is often useful to persist data over different sessions, and as JavaScript variables are cleared when refreshing the page, other methods for storing data are needed. A common use case for persistent storage methods is keeping track of users by

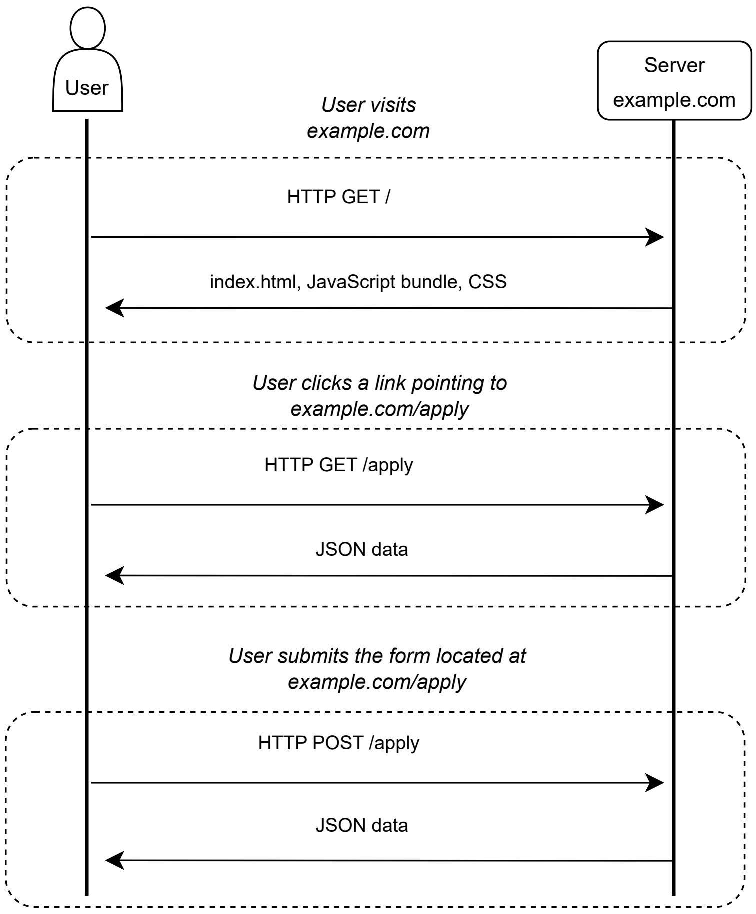


Figure 3: Network traffic for a user interacting with a Single Page Application.

storing a session token in the client browser, allowing authenticated users to navigate a web page without having to authenticate every time they visit a new page.

2.3.1 Cookies

Cookies have long been used to allow servers to store state at HTTP user agents, such as web browsers, over the HTTP protocol, which itself is mostly stateless. Cookies function by passing data in HTTP headers. By using the Set-Cookie field, servers can pass name/value pairs and related metadata to user agents. In later requests, the user agent can set a previously stored value in the Cookie field, allowing the server to identify the client or to keep track of user preferences. Cookies are not always suitable for keeping track of user state or identifiers. Using cookies, a server cannot differentiate between different sessions for one user. As an example, if a user had two separate windows open at the same time and was trying to purchase two different items, the session would leak between the two windows [18]. Cookies are also limited by their max size, which is 4096 bytes.

When setting cookies, the server can define the scope of the cookie, which can include the domain and path of the cookie, as well as if it should be allowed to be used in HTTP requests that do not use TLS [19]. The use of scope for cookies is essential if it contains confidential information such as session identifiers. If a domain is not defined for the cookie, the value could be read by any visited website, significantly increasing the chance for session hijacking. Similarly, if the cookie is allowed to be used without TLS, any network attacker would be able to read the transmitted data. To limit many vulnerabilities, such as cross-site scripting (XSS) attacks, the *HttpOnly* flag can be set for cookies that are used as session identifiers. The flag blocks access to the cookie from JavaScript executed in the client, only allowing it to be sent as part of a request in the Cookie header.

To further protect against cross-site attacks, the *SameSite* attribute can be used [20]. Three different *SameSite* policies exist: None, Lax and Strict. A cookie with the None policy is attached to all outgoing requests, which includes cross-site requests. Cookies with the Lax policy are attached to same-site requests as well as cross-site requests with safe HTTP methods, which includes using the HTTP GET method and the request resulting from a top-level navigation by the user, such as clicking on a link. Cookies with the Strict policy are only included on requests that originated from the same origin. As such, if a user clicks a link to a site on a page with some other origin, the cookie will not be included.

2.3.2 Web storage API

The web storage API was created to deal with some of the issues of cookies. The API includes two mechanisms for storing data, session storage and local storage [18]. Both storage methods allow persisting data for longer periods of time, while also separating the storage by domain. As data is unencrypted at rest, the web storage API is inherently insecure. Additionally, web storage is vulnerable to a number of attacks, such as XSS attacks, requiring developers to implement additional security features and considering what kind of data to store. Due to these security concerns, web storage should not be used to store any sensitive data. Session storage persists data for the lifetime of the session, keeping the data through page navigations and refreshes until the browser or tab is closed [21]. Local storage persists data over multiple sessions, having to be explicitly cleared by using JavaScript or by clearing all browser storage.

2.4 Authenticating HTTP requests

As internet-facing applications can be accessed by anyone, it is often necessary to authenticate entities trying to communicate with the application. Authentication can be used for limiting access to only certain users or to display user-specific information. As the authenticated entities can vary from human users to devices or other servers, different ways of authentication can be appropriate for different systems. In a traditional web application with a client, a backend and a database, the authenticated parties are typically the client when communicating with the backend and the backend when communicating with the database.

The need for authenticating HTTP requests has existed since the early days of the internet, with the basic HTTP authentication scheme being an early method [22]. Basic authentication functions by sending an authorization header with a base64-encoded string containing a username and password, which the server can use to identify the user, by comparing the string with a stored value, which should be encrypted at rest. Notably, the scheme does not provide any confidentiality for credentials, as the credentials are not encrypted or hashed in any way, allowing for the base64 value to be easily decoded. While the encryption of headers in TLS prevents outside attackers in the network from reading the credentials, this lack of confidentiality allows the receiving server to access the plain text credentials. This allows hostile or compromised servers to capture credentials which can be used on other sites in the case of credential reuse. A 2014 study by Das et al. found that 43% of users directly reuse passwords on different sites [23]. With these issues and the prevalence of password reuse in mind, the basic HTTP scheme can no longer be considered appropriate for many web applications.

Human-readable passwords are not always required, and in many cases API keys can be appropriate for authenticating clients [24]. API keys differ from basic authentication by only identifying users by a single secret string that is shared between the client and server, instead of a string and a username. As developers can define the authentication string freely, it could in theory be constructed similarly to the string used in basic authentication. In practice, the API key is often a completely random string, with the server keeping a track of who the string belongs to if required. The API key can also be a hashed string similar to the one used in basic authentication, allowing the storage of a user identifier and password without the risk of leaking either the password or the user identifier. Similarly to basic authentication, API keys are sent in a request header, often using the X-API-KEY header. The API key could also be sent as a query parameter or in the request body itself depending on the implementation. API keys are especially useful if the server does not need to differentiate between clients, if there are a small number of clients or if the client is not a user-facing web client, such as a backend server communicating with an external API. While API keys mitigate some of the risks associated with sending unencrypted passwords over the internet, they suffer from many of the same flaws as basic authentication. Similarly to basic authentication, the API key is the same for each request and as such it only has to be leaked once for attackers to gain unapproved access. While systematically rotating API keys can lower the risk of the API keys leaking, they are not optimal for all use cases today.

Both of the described authentication schemes share the weaknesses of requiring servers to store the user identifiers and reusing the same identifier for an unspecified amount of time. Servers also have to implement the authentication logic separately, leaving room for poor implementations and suboptimal credential storage. Token-based authentication has become a popular way to outsource the authentication logic to a trusted third party. Token-based authentication functions by redirecting the user to an identity provider (IdP) which authenticates the user and grants a token identifying the user [25]. A token can be a random string with sufficient length to make it unlikely for a third party to be able to guess it. A token can contain some information about the user, but it can also be completely opaque. The client can store this authentication token and send it with any future request, typically as a cookie (described in section 2.3.1), allowing servers to verify the user by querying the IdP with the user token. Compared to the static API key, authentication tokens are typically only valid for a short amount of time, mitigating the risk of credential hijacking. To remove the need for users to log in continuously, access tokens can often be refreshed silently in the background by the client application.

2.5 Cross-site request forgery

Cross-site request forgery (CSRF) is an attack where an attacker causes a victim's browser to perform an unwanted operation on a trusted website. CSRF attacks rely on the fact that stored user sessions belonging to a specific domain tend to be included automatically in all HTTP requests to the domain. As such, if the victim is tricked to click a link to the target site with some malicious payload, such as a form submission to update a password, the victim's browser will automatically include the victim's session in the request, allowing the attacker to perform operations as the victim. A typical CSRF attack has the goal of gaining access to resources belonging to the victim [26]. This can give attackers access to private resources belonging to the victim or allow attackers to impersonate the victim. By forging a request that signs the victim in as the attacker to a legitimate site, a victim can be tricked to perform operations as the attacker in [27]. These login CSRF attacks are more difficult to defend against than traditional CSRF attacks but can be equally damaging. An attacker could for example trick the victim to store a payment method in the attackers account.

CSRF attacks can be performed using a number of methods that all rely on somehow causing the victim's browser to visit a link provided by the attacker. In an ideal case for the attacker, the vulnerable route is a HTTP form that allows the attacker to perform an operation as the victim, such as transferring money or changing account passwords. On a website with user-generated content, such as a forum, attackers could embed malicious links in images. For malicious images to be executed on page load the site will need to have a vulnerable route where a HTTP GET request executes an operation with side-effects. According to the HTTP specification, GET requests should be free of side effects, but it is up to developers to correctly implement the specification. If a victim visits a malicious site controlled by the attacker, the malicious site can instruct the user's browser to execute any HTTP request to the vulnerable site. Compared to the malicious embedded images, the target site does not need to have any wrongly configured routes that execute GET requests with side-effects, as the attacker can freely choose the type of HTTP request.

As the potential for CSRF attacks have been known for many years, a number of well-known defenses exist. Bart et al. (2008) suggest protecting against CSRF attacks by using secret validation tokens [27]. A secret validation token is a random value that is included by applications in each HTTP request to ensure that requests have been sent from an authorized source. The token should be hard to guess for attackers that do not already have access to the victim's account. As the token will have to be stored in the client, end users will be able to access it with ease in many cases. As such, the token should never be shared between user or sessions, and should ideally be treated as a single-use nonce to make replay attacks less likely.

3 The OAuth 2.0 specification

OAuth is an authorization framework that implements token-based authentication. In the context of OAuth, it is necessary to clarify the difference between authentication and authorization. Authorization is the process of determining whether an entity can access a resource. Authentication on the other hand aims to identify an entity. Authentication is a prerequisite for authorization, as an entity has to be securely identified for access control to be relevant [28].

Version 1.0 of the specification was released in 2010 [29], and the current latest version, 2.0, was released in 2012 [25]. As version 1.0 of the spec is obsolete, this thesis will only consider OAuth 2.0. This section gives an overview of the specification, describing the different roles and identifiers as well as different grant types, with a focus on the authorization code grant. The OAuth specification defines four separate flows: the authorization code grant and implicit grant that are used to get a new access token, the refresh flow that is used to refresh an existing access token and the introspection request which is used to get information about an existing access token.

3.1 Roles

The abstract authorization flow of the OAuth specification defines four separate roles:

- **Resource owner**
  The resource owner is an entity that can grant access to a protected resource. Can be referred to as an end user if it is a person.

- **Resource server**
  The resource server hosts the protected resource and can accept and respond to access requests using access tokens.

- **Client**
  The client is an application requesting a protected resource on behalf of the resource owner. The client can be implemented freely and can be run on any type of device. The OAuth specification defines two types of clients, confidential and public. Confidential clients are clients that can securely store their credentials, such as clients implemented on secure servers with restricted access. Public clients are clients that are unable to securely store their credentials, such as clients executing in web-browsers or natively on the end user's device. The identity of the client should be verified by the authorization server where possible. The client credentials should be kept secret to avoid malicious clients impersonating the client. If client authentication is not possible, the authorization server should only allow redirection to specific URIs, which can prevent delivering tokens to counterfeit clients. Public clients are required to define their allowed redirect URIs. Confidential clients should also define their allowed redirect URIs but can choose not to.

- **Authorization server**
  The server issues access tokens to the client after authenticating the resource owner. The authorization server can be the same entity as the resource server, but the interaction between these entities is not defined in the specification. The authorization server must use TLS for all requests sent to the authorization or token endpoints to prevent man-in-the-middle attacks. The client is required to validate the authorization server's TLS certificate. The authorization server can also be referred to as a provider.

3.2 Identifiers

The OAuth specification utilizes a number of tokens and other identifiers that are used to identify resource owners and clients.

- **Authorization code**
  The authorization code is granted by an authorization server to a client after the server has successfully authenticated the resource owner. The authorization code is a random string, and it does not typically contain any data in itself. The authorization code is used by the client to request an access token from the authorization server. As the authorization code should only be used once to request an access token, it must only be valid for one request and must be short-lived. If an authorization code is used multiple times, the authorization server should revoke all tokens that have been generated using the code. In the implicit code grant, an authorization code is not used, with the authorization server issuing an access token directly instead.

- **Access token**
  Access tokens are strings which are used to grant access for a specific client to a specific resource server. The token additionally includes a scope for the granted access, making it possible to grant granular access to resources. To mitigate the chance for leaked tokens being abused, the token can be given an expiration time. Access tokens and related attributes have to be kept confidential in transit and storage. Access tokens should only be shared between the client it is issued to, the authorization server that creates the token and the resource server which the token is valid for. It must not be possible for unauthorized parties to generate, modify or guess valid access tokens.

- **Refresh token**
  Refresh tokens are strings which are used to request additional access tokens. Refresh tokens are created simultaneously with access tokens, and are useful when the access token has an expiry time set, allowing for end-users to stay authenticated for a longer period of time if the client stays active by refreshing new access tokens before the previous token expires. The refresh logic is typically performed in the background by the client, causing minimal interruption for end-users. When granting a new access token from a refresh token, the authorization server can issue a new refresh token, allowing for multiple refreshes for one

authorization. Refresh tokens and related attributes have to be kept confidential in transit and storage. Refresh tokens should only be shared between the client it is issued to, the authorization server that creates the token and the resource server which the token is valid for.

3.3 Authorization flows

This section describes the two authorization flows in the OAuth specification, the authorization code grant and the implicit grant. Due to the weaker security properties of the implicit grant, the authorization code grant should be preferred over the implicit grant where possible. As such, the main focus of this section is on the authorization code grant.

3.3.1 Authorization code grant

The OAuth specification defines an authorization code grant, which can be used to obtain access tokens and refresh existing tokens. The authorization flow is based on redirections. Parameters are passed as query parameters to the authorization server using the *application/x-www-form-urlencoded* format [30]. A typical authorization code grant, described in figure 4, functions as follows [25]:

1. **Initiate authorization**
   The resource owner, or end user, initiates the authorization process. This could be a user clicking a login button, a user trying to perform an action requiring authorization or a user wanting to give access to some of its information to a third party.

2. **Redirect end user to authorization endpoint**
   The client handles the authorization request by directing the resource owner to the authorization server. The client passes the client identifier, the requested scope and can optionally pass the local state to the authorization server. The state is an opaque value, such as a hash of a unique user session identifier, that is used by the client to maintain state between the request and the callback. State is used to protect clients against Cross-site request forgery (CSRF) attacks, where the user-agent of a victim is made to follow a malicious URI to a trusting server, allowing attackers to inject their own authorization code to the request, which can cause victims to be signed in as the attacker or can even give the attackers access to protected resources belonging to the victim. A redirection URI is also included, to which the authorization server should redirect the end user to once the authorization process is complete.

3. **Authenticate user**
   The authorization server authenticates the end user. The OAuth specification does not define how this authentication is implemented, leaving it up to each authorization server. Typical authentication methods are traditional username

and password or biometrics and can include multi-factor authentication methods (MFA). The authorization server requires the end user to confirm the authorization to the specific client, preventing common CSRF attacks.

4. **Authorization response**
If the end user grants access and is successfully authenticated, the authorization server responds with an authorization response. The response contains an authorization code and the client state, if it was passed in the initial authorization request. The response is sent to the redirect URI that was sent in the authorization request.

5. **Request access token**
After obtaining an authorization code, the client requests an access token from the authorization server. The request includes the authorization code, in addition to a client identifier and redirect URI. The request can also include a scope for the token, which the authorization server takes into account when generating the token, limiting what the token can be used for.

6. **Access token response**
The authorization server confirms that the authorization code is valid and authorized, and ensures that the code was issued to the client that is making the token request. After validating the code, the server responds with an access token, an expiration time and an optional refresh token. The scope of the access token should be the same as requested in the token request, if a scope was included. The authorization server takes the client identity into account when deciding the token scope, making it possible for the token to be given less rights than was requested.

7. **Request protected resource**
The client can now use the access token to access protected resources by adding the token to the request body, to the request headers or as a query parameter, depending on the resource server. This request can be repeated by the client as long as the access token is still valid.

8. **Introspection request**
When receiving a request for a protected resource, the resource server has to validate the provided access token. How this token validation should be implemented is not defined in the original OAuth 2.0 specification, but it is defined in a separate OAuth 2.0 Token Introspection specification [31]. The only required parameter for the token introspection request is the access token itself, with the option to send additional parameters, such as a token type hint.

9. **Introspection response**
The authorization server responds with a JSON object containing information about the token, with a boolean value indicating if the token is active as well as a unique identifier, *sub*, as the only required parameters. The response can also contain a number of parameters describing the token, such as an identifier

for the client it was issued, the username of the resource owner, the scope of the token or the token type. The response can also include an expiry time, a timestamp for when the token was originally issued, a timestamp for when the token becomes active, a subject for the token, the intended audience for the token, the issuer of the token as well as a string identifier for the token.

10. **Return the requested resource**
    Based on the parameters returned by the authorization server, the resource server can return the protected resource, perform an action requiring authorization or return some personalized data.

3.3.2 Implicit grant

Access tokens can also be obtained using the implicit grant type, which simplifies the authorization flow of the authorization code grant by removing the issuing of authorization codes. This shortens the number of round-trips required, improving the responsiveness of interactive applications. The steps of the implicit grant are similar to the authorization code grant described in figure 4, with the authorization server responding with an access token directly, instead of issuing an authorization code first. The implicit grant type removes the use of refresh tokens, requiring end users to authenticate multiple times if a session takes longer than the access token is valid. As the access token is included in the response URI, there is an increased chance for the token leaking. The implicit grant does not provide a method for identifying clients, making it impossible for authorization servers to validate that the used token was requested to the client using it.

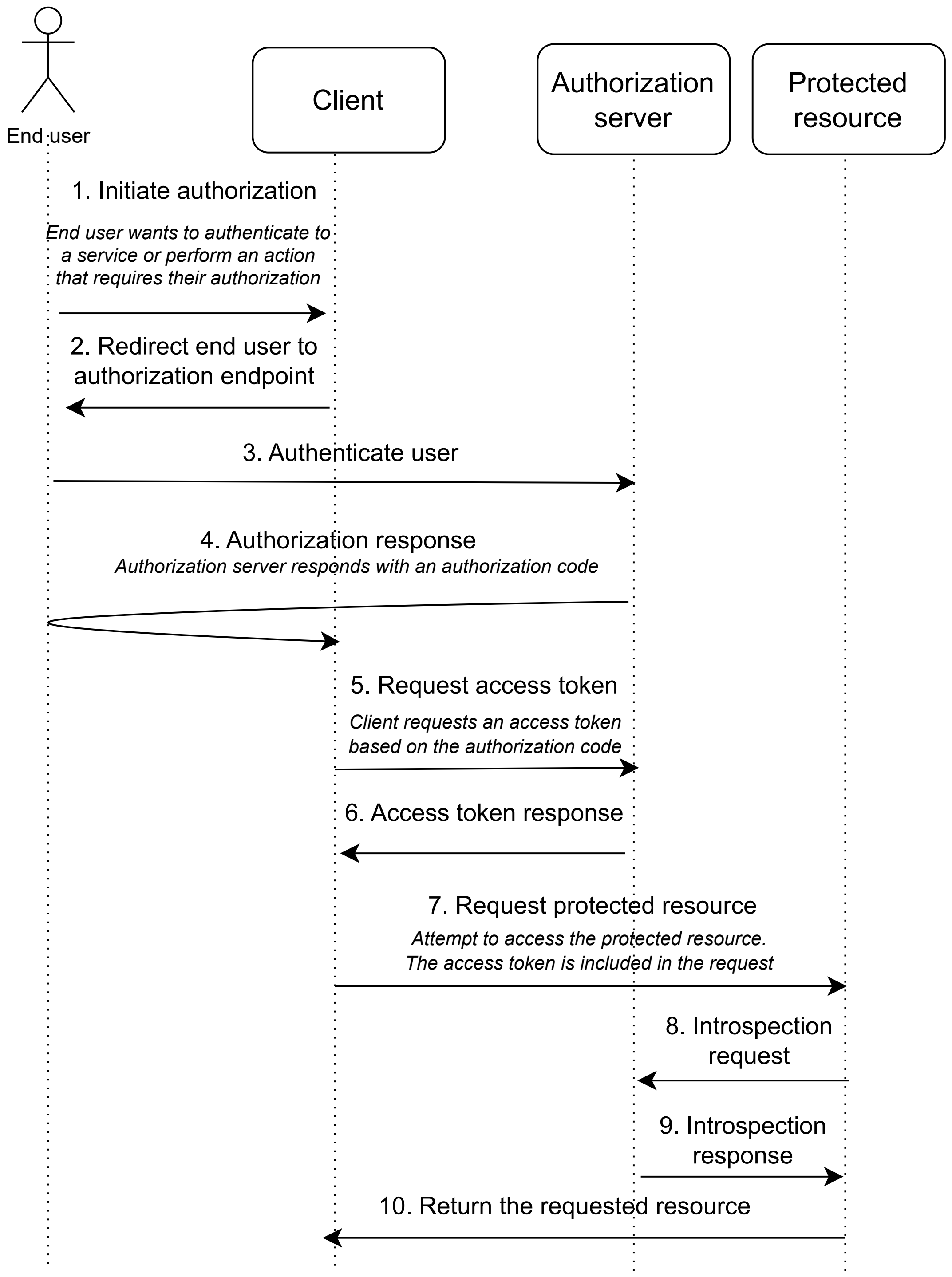

End user
Client
Authorization server
Protected resource
1. Initiate authorization
End user wants to authenticate to a service or perform an action that requires their authorization
2. Redirect end user to authorization endpoint
3. Authenticate user
4. Authorization response
Authorization server responds with an authorization code
5. Request access token
Client requests an access token based on the authorization code
6. Access token response
7. Request protected resource
Attempt to access the protected resource. The access token is included in the request
8. Introspection request
9. Introspection response
10. Return the requested resource

Figure 4: Accessing a protected resource using OAuth 2.0 authorization code grant.

4 Vulnerabilities in OAuth systems

The OAuth protocol does not strictly define how systems utilizing the protocol should be implemented, requiring developers to thoroughly consider the security of their implementation choices. When inspecting the security of OAuth systems, the critical components are the client and the authorization server. Poorly implemented OAuth systems are very common, with Philippaerts et al. (2022) finding some vulnerability in each inspected authorization server [32]. Client implementations are not any more secure, with Yang et al. (2016) finding simple mistakes in a majority of inspected web applications [33]. As these vulnerabilities are obvious enough to be found with automated methods, an attacker is very likely to find them as well.

4.1 Vulnerabilities in authorization servers

As the authorization server is a trusted entity in the OAuth authorization flows, it is essential that it is securely implemented. In addition to following general best practices for server management, such as keeping systems up to date, developers should consider OAuth-specific vulnerabilities. To avoid on-path attackers, TLS should be used for all communication with the authorization server. To lower the chance of developer mistakes, clients should be forced to use TLS for all communication with the authorization server.

4.1.1 HTTP 307 redirect

In the redirection-based authentication flows of the OAuth specification, how the redirections are implemented can have a significant impact on the overall security of the system. While the choice of HTTP status code used for redirection is seen as an implementation detail in the OAuth specification, the status code used should be chosen carefully. In contrast with other types of redirects, if the 307 HTTP status code is used as redirection, the body of the original POST request can be redirected. If the authorization server uses HTTP 307 to redirect users to the client after authenticating, the client could receive the credentials that were originally sent in the body of the HTTP request to the authorization server. If an attacker tricks a victim to authenticate to the legitimate authorization server through a malicious client, the attacker could receive the credentials [2].

The 307 redirect attack is similar to a traditional phishing attack, where a victim is tricked to sign in to a malicious website, giving the attacker their credentials [34]. The 307 redirect attack is more likely to succeed compared to a traditional phishing attack, as it can be much harder to distinguish from a legitimate authentication. In the attack, the victim authenticates to the legitimate authorization server, making it difficult to distinguish from a normal authentication. Additionally, tricking the victim to visit the malicious client could be achieved in one single click, without the victim visiting the malicious site at any point, instead directly being directed to the legitimate authorization server. As such, the victim does not have to miss subtle details in the login form that are slightly off, or mistake the malicious domain for a legitimate one as in traditional phishing attacks.

To clarify that the request body should not be sent together with the redirection, the HTTP code 303 should be used, as it explicitly states that the body should not be included in the response [35]. Using a correct HTTP code does not solve the issue of sending credentials together with the client callback, as the authorization server can choose what content to send regardless of the status code used. While correctly dropping the request body will have to be implemented by the authorization server, by using a HTTP status code that explicitly states that the request body should be dropped, the likelihood for faulty implementations is lower, compared to the 307 code which explicitly states that the original request body should be included in the redirect.

4.1.2 Insecure token handling

Due to the central role of tokens in OAuth authorization flows, their secure handling is essential. In the authorization code flow the relevant tokens are access tokens, refresh tokens and authorization codes. To avoid leaked tokens being abused, each token should only be valid for a short amount of time. To extend the lifetime of user sessions and their corresponding access tokens, refresh tokens should be used.

In the authorization code grant, authorization codes are used to request an access token from the authorization server. In a correct authorization code flow, the code should only be used once, and it should be used quickly after it has been granted. As such, to avoid replay attacks, the authorization server should only be valid for one request, and it should have a short expiry time [36]. Additionally, if an authorization code is used multiple times, the authorization server should invalidate all related codes and tokens. As an authorization code should never be used more than once in a correct code flow, duplicate use implies that something has gone wrong in the client or that an attacker is attempting a replay attack.

Refresh tokens should be dealt with similarly to authorization codes, being single use and only valid for a set amount of time. The expiration time for refresh tokens can be longer than that of authorization codes, and can vary based on the specific client. While the OAuth specification is quite clear in requiring the rotation of refresh tokens, Philippaerts et al. (2022) found that 44% of authorization servers accept old refresh tokens that have already been used [32]. The old refresh tokens should be stored, and all related tokens should be invalidated if an old refresh token is used. Compared to storing one old authorization code to detect replay attacks, detecting old refresh tokens can require storing a significant amount of tokens. The large number of stored refresh tokens can require large amounts of storage, and the comparison to old refresh tokens can cause significant storage load [37]. As such, developers should weigh the benefits of storing old refresh tokens with the costs when deciding how long old tokens should be stored for.

4.1.3 Redirect URI attack

As the generated authorization code and refresh and access tokens are sent to the redirection URI, the authorization server has to validate that the redirect URI is correct. The OAuth specification states that clients should register each allowed redirection URI, and that the authorization server should check the complete URI. In practice, the redirect URI validation can be implemented in a number of ways. While the minimum check should be to compare the complete redirect URI against a stored value, many authorization servers only check the origin of the redirect URI, allowing an attacker to change the path of the redirect URI [38]. For increased security, authorization servers can also generate client-specific redirect paths when a client registers.

In the authorization code grant, the client sends the received authorization code to the authorization server to receive an access token. Clients typically define a separate redirect URI for each provider, such as */oauth/callback/google* and */oauth/callback-/amazon*. If an attacker is able to tamper with the path of the redirection URI, with methods such as malicious images or iframes, the client could be tricked to send the code to the wrong provider [38]. If the client application has an XSS vulnerability at a certain path, an attacker could set that vulnerable path as the redirect URI. While OAuth tokens are generally secure against XSS as long as they are stored as HttpOnly cookies, if an attacker is able to execute code in the redirect flow they will be able to read the code or token that has been generated, as they are sent as query parameters to the client before being stored as a cookie. This allows attackers to impersonate the victim, or to access protected resources belonging to the victim.

4.2 Vulnerabilities in OAuth clients

When implementing OAuth-based authentication systems, a developer can choose to rely on external authorization servers, greatly simplifying the authentication logic they need to implement on their own. When relying on external authorization servers, a developer is limited to studying the available providers and choosing ones that are suitable and secure. This in turn allows developers to focus on implementing a secure client. The most common client vulnerabilities are related to poor CSRF protections due to misuse of the *state* parameter, poor tracking of user intent and incorrect token storage in the client.

4.2.1 Invalid state handling

When used correctly, the state parameter provides protection against CSRF attacks [39], with the OAuth 2.0 specification stating that the state parameters should be used for all authorization requests. However, a 2016 study by Yang et al. found that 61% of studied applications did not use the state parameter, while 55% of applications that use state mishandled it in some way [33]. A study carried out by Almgren et al. (2015) was less pessimistic, but still found that 25% of the Alexa Top 10 000 websites did not implement standard CSRF protections, such as correctly utilizing state [39].

The OAuth authorization flows utilize redirects to provide access tokens and authorization codes to clients. If the state parameter is not used by the OAuth client, clients have no way to verify how an authorization flow was initiated. As such, if the state parameter is missing, many types of CSRF attacks become easier to execute for attackers. The simplest form of CSRF attack that can be performed on clients that do not use the state value is the login CSRF attack. In the login CSRF attack, a victim is tricked to authenticate to a legitimate site using the legitimate authorization server, but signing in the victim as the attacker. This can lead to the victim performing operations as the attacker, and if the victim stores any information while they are logged in, such as payment information, the attacker will gain access to them.

Missing state validation can also be used to attack the federation process of an OAuth system that is used to sign in users [40]. The attack targets the account creation process, causing the victim's account on a relying site to bind to an account at the provider that is controlled by the attacker. While the attack is similar to the login CSRF attack, it provides more persistent access to attackers, and has an increased probability of leaking the victim's personal information, as such information is typically provided when creating an account.

While state is used by many clients, it is not very useful if it is not implemented correctly. Similar attacks can often be performed on clients mishandling the state parameter as on those that do not consider the parameter at all. The state parameter should be random and difficult to guess, should ideally be single-use and should under no circumstances be shared between sessions or users. Yang et al. (2016) identified a number of common mistakes in state validation [33]:

- **Lack of state validation**
  The state parameter is used but its value is never actually validated by the client

- **Lenient state validation**
  Some clients only validate the state parameter if it is provided. However, they do accept requests that are missing the state parameter.

- **State not bound to user**
  The client assumes that all state parameters that it has generated are valid but does not check which user the state parameter belongs to. This allows an attacker to perform a CSRF attack by substituting the victim's state with their own.

- **State not single use**
  If the state parameter is not single use, an attacker can use a previous parameter in a replay attack. The parameter could be obtained by eavesdropping or some other form of leaking. The risk for leaking the state parameter is especially high if TLS is not used. Some common characteristics of state reuse are keeping the same state value until the user tries to log in again, keeping the same state value as long as the user uses the same browser, or using a constant state value across all sessions and users.

4.2.2 IdP mix-up attack

If the OAuth client fails to keep track of the authorization server that the user has chosen, an IdP mix-up attack could be performed by a network attacker [2]. In the IdP mix-up attack, a victim's authorization code or access token is sent to an authorization server controlled by an attacker. To perform the attack, the OAuth client must allow multiple different authorization servers, and one of the allowed servers must be malicious. In practice, this could occur if one authorization was compromised by attackers, providing them further access. An attacker must also be able to modify requests sent to the OAuth client, which could be accomplished by an active network attacker.

The IdP mix-up attack in an authorization code flow is described in figure 5. The request initiating authorization (request 1) is modified by an attacker, changing the chosen authorization server from the honest authorization server to a malicious server controlled by the attacker. The client stores the choice of authorization server locally, and redirects the user agent to the malicious authorization server (requests 2 and 3). The malicious server redirects the user agent to the honest authorization server, making it very difficult for the victim to notice anything out of the ordinary (request 4). The end user authenticates normally to the honest authorization server, and the authorization server sends a response containing an authorization code to the OAuth callback route of the client (request 5). As the client believes that the malicious authorization server should be used, it sends the authorization code to the malicious authorization server when it attempts to request an access token (request 6). The malicious authorization server now has the victim's authorization code, and can request an access token (requests 7 and 8). Having gained access to the access token, the attacker can now impersonate the victim or gain access to protected resources belonging to the victim [41].

4.2.3 Vulnerabilities in token storage

When storing data in web clients, the user will be able to access the data, no matter how much a developer attempts to secure it. Client-side storage such as cookies or local storage are also vulnerable to XSS attacks. As such, if the goal is to maximize security, public clients should completely avoid persistent local token storage.

Even if persistent local storage is avoided, sensitive codes and tokens will have to be temporarily stored in JavaScript. Developers should consider how this temporary storage is implemented. One possible hardening option is using JavaScript closures. When using closures, the variables inside the closure can only be accessed by code inside the definition area of the closure, significantly reducing the chance that a XSS attack would succeed in gaining access to client secrets [42]. Browsers also allow running code in web workers, which act as a separate thread. In addition to allowing concurrent programming, web workers allow executing JavaScript code in an isolated sandbox. Similarly to closures, this greatly reduces the risk of credentials leaking in case of an XSS attack, as the malicious code would have to be executed in the web worker containing the secret to be able to access it [43].

While persistent token storage has security issues, the added benefits provided by users being able to stay signed in across multiple sessions and browser restarts can outweigh the costs. When persisting tokens, developers are limited to using the web storage API and storing session data in cookies. Data stored using the web storage API and as cookies are limited to a specific domain, making it impossible for malicious

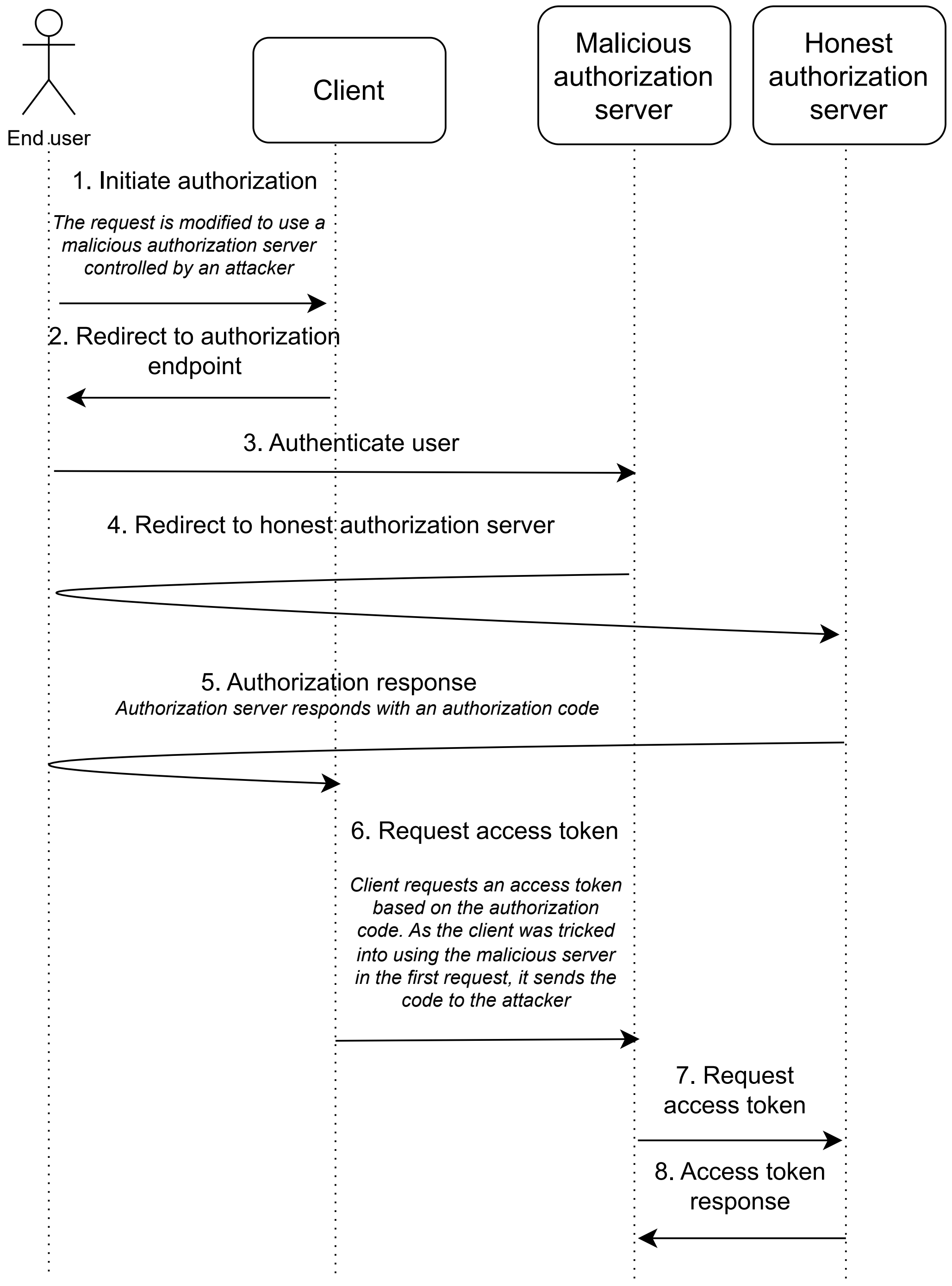

End user
Client
Malicious authorization server
Honest authorization server
1. Initiate authorization
The request is modified to use a malicious authorization server controlled by an attacker
2. Redirect to authorization endpoint
3. Authenticate user
4. Redirect to honest authorization server
5. Authorization response
Authorization server responds with an authorization code
6. Request access token
Client requests an access token based on the authorization code. As the client was tricked into using the malicious server in the first request, it sends the code to the attacker
7. Request access token
8. Access token response

Figure 5: IdP mix-up attack in an authorization code flow

sites to directly access the sensitive data. Both methods are however vulnerable to XSS attacks, as they can be accessed using JavaScript. When using cookies, XSS attacks can be mitigated by setting the *HttpOnly* flag. When the flag is set, the cookie can no longer be accessed by arbitrary JavaScript, with the cookie being included in requests to a specific domain. The web storage API does not provide any protections similar to *HttpOnly* cookies, and as such it should be avoided when storing any sensitive information. As the cookie value is included in every request, this leaves additional room for potential CSRF attacks, making other types of CSRF protections, such as correctly utilizing the *state* parameter even more critical.

5 Implementing an OAuth client

This chapter describes a practical implementation for an OAuth 2.0 client using an authorization server provided by a third party. The central design choices are described in addition to their effect on the security of the system. The OAuth client consists of a minimal frontend application that interacts with an API backend which handles all communication with the authorization server. The backend supports using multiple authorization servers simultaneously, with the example implementing authorization using Google and Microsoft accounts. The focus of the chapter lies on the authorization code flow, as the process of retrieving an access token is the most critical part of an OAuth system. It should also be possible to implement the authorization code flow in a similar way as in the example for a wide variety of different applications. The chapter briefly describes token introspection and token refreshing, but as their implementation will vary greatly between different applications, they are not described in detail. The complete source code for the client can be found on GitHub [44].

5.1 Application structure

When implementing a modern web application, developers are left with a number of choices. As described in section 2.2, the utilization of JavaScript allows developers to implement complete applications that are fully contained on the end user's machine. When implementing an OAuth client, it is possible to contain all logic on the client machine. This does however negatively affect the overall security of the system. Single-page applications (SPAs), where all logic is contained in the browser, are considered public OAuth clients, as a user can access and study every part of the system. In such clients, the authorization code flow can only be partially implemented, as it relies on the client proving its identity using a client secret. As the client is completely stored in the user's machine, a malicious user can simply analyze the application and retrieve the secret value. As the benefits provided by the authorization code flow are not fully realized in public clients, they often utilize the implicit grant instead, which is inherently less secure than the authorization code flow.

Due to these limitations of pure SPAs, the example implementation splits the OAuth client into a frontend and a separate API backend. The user interacts with the frontend which in turn interacts with the API. The API is responsible for all communication with the authorization server, allowing the client to securely store client secrets. The separate API also makes it possible to more securely store access tokens and other secrets belonging to the end user. These storage options are described in detail in sections 5.2 - 5.3.

5.1.1 Obtaining an access token

The OAuth client implements the authorization code grant to authenticate users. As the client is separated into two parts, some additional steps are required compared to a minimal authorization code grant. The network traffic of the authorization flow is shown in figure 6. Each step in the authorization flow is detailed below, with code snippets describing how they are implemented.

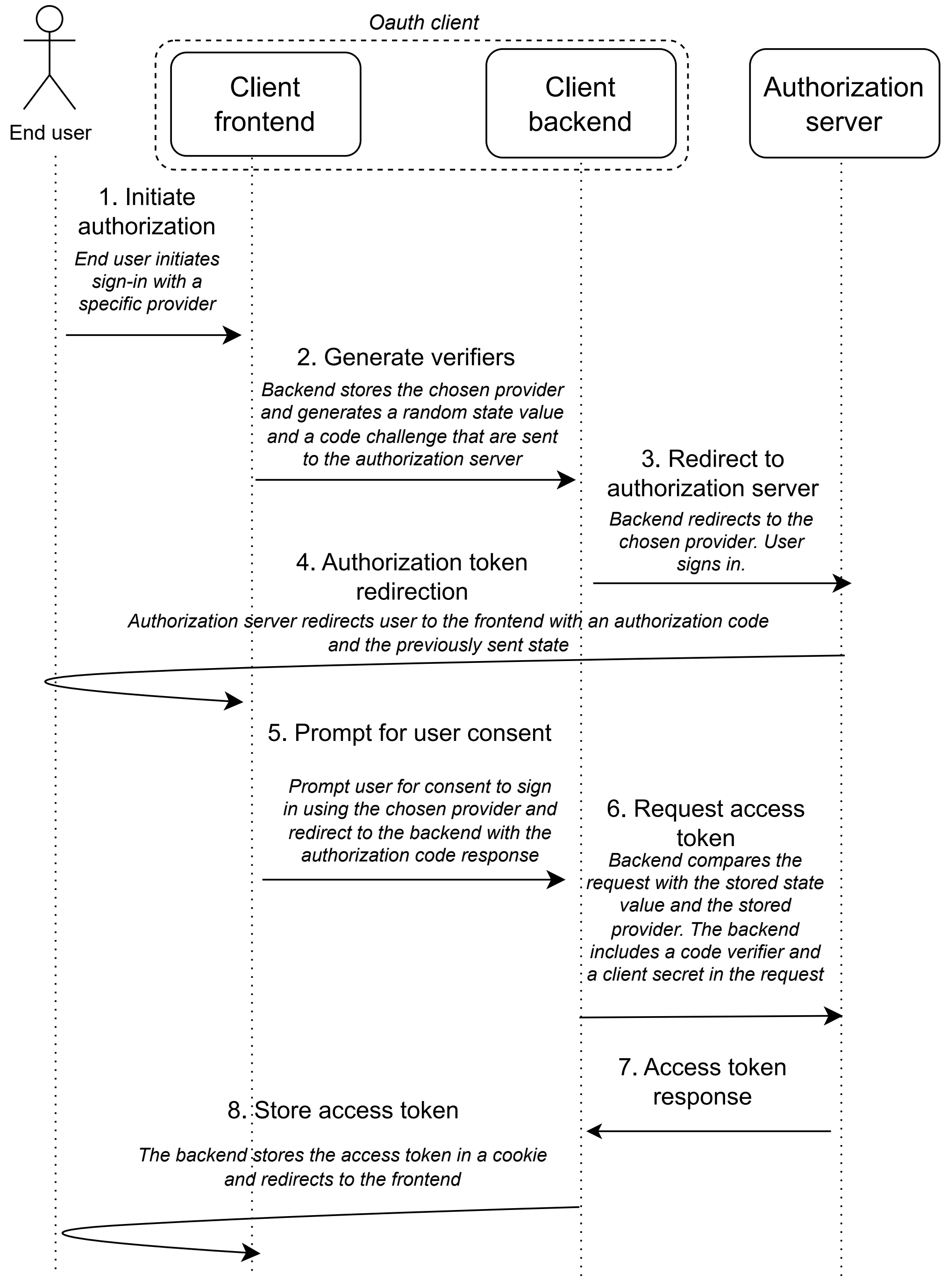

Oauth client
End user
Client frontend
Client backend
Authorization server
1. Initiate authorization
End user initiates sign-in with a specific provider
2. Generate verifiers
Backend stores the chosen provider and generates a random state value and a code challenge that are sent to the authorization server
3. Redirect to authorization server
Backend redirects to the chosen provider. User signs in.
4. Authorization token redirection
Authorization server redirects user to the frontend with an authorization code and the previously sent state
5. Prompt for user consent
Prompt user for consent to sign in using the chosen provider and redirect to the backend with the authorization code response
6. Request access token
Backend compares the request with the stored state value and the stored provider. The backend includes a code verifier and a client secret in the request
7. Access token response
8. Store access token
The backend stores the access token in a cookie and redirects to the frontend

Figure 6: Network traffic of the OAuth clients authentication flow

1. **Initiate authorization**
   The user begins the sign-in process by clicking on a button to sign in with a specific provider.

```
const LoginButton = ({ provider }: { provider: string }) =>
    {
  return (
    <button
      onClick={() =>
        (window.location.href = `${API_URL}/oauth/login/${
   provider}`)
      }
    >
      Login with {provider}
    </button>
  )
}
```

Listing 1: Login button

2. **Generate verifiers**
   The user is redirected to the client backend to a route belonging to the chosen provider. The backend server generates two random 32 byte strings, with one string used as a state value and the other string used as a code verifier. A code challenge is created based on the code verifier to be used by PKCE, described in more detail in section 5.3. The backend stores the code verifier, the state value as well as the chosen provider as cookies in the user's browser.
   The code challenge is generated with the following function:

```
export const generateCodeChallenge = (codeVerifier: string)
    => {
  return base64url(crypto.createHash('sha256').update(
   codeVerifier).digest())
}
```

Listing 2: Code challenge generation

3. **Redirect to authorization server**
   The user is redirected to the chosen authorization server. The state and code challenge generated in the previous step are included as query parameters, in addition to a *redirect URI*, to which the authorization server will redirect the user after successfully authenticating. A client id provided by the authorization server is also included, as well as the *access_type* which indicates whether a refresh token should be included.
   The URL to the authorization server is constructed with the following function:

```
export const getAuthServer = async (
  state: string,
  codeChallenge: string,
  provider: string
) => {
  const endpoint = await getEndpoint('authorization_endpoint', provider)
  const oauthCredentials = config.providers[provider]
  if (endpoint === undefined || oauthCredentials === undefined) {
    return undefined
  }

  // Refresh token should be requested as part of the scope
  // for Microsoft, while it is requested from other
  // providers with access_type='offline'
  const scope = provider === 'microsoft' ? 'offline_access openid' : 'openid'

  return (
    endpoint +
    '?response_type=code' +
    `&client_id=${oauthCredentials.clientId}` +
    `&scope=${scope}` +
    `&redirect_uri=${config.redirectUri}/${provider}` +
    '&access_type=offline' +
    `&state=${state}` +
    `&code_challenge=${codeChallenge}` +
    '&code_challenge_method=S256'
  )
}
```

Listing 3: Constructing the authorization server URL

The complete login route is implemented as follows, where *oauthRouter* is an express router [45]:

```
oauthRouter.get('/login/:provider', async (req: Request, res: Response) => {
  const { provider } = req.params
  const state = randomBytes(32).toString('hex')
  const codeVerifier = base64url(randomBytes(32))
  const codeChallenge = generateCodeChallenge(codeVerifier)

  const authServer = await getAuthServer(state, codeChallenge, provider)
  if (authServer === undefined) {
    return res.sendStatus(500)
  }

  res.cookie('state', state, {
    httpOnly: true,
    secure: true,
    sameSite: 'lax',
    signed: true,
    maxAge: 60 * 1000,
  })

  res.cookie('code_verifier', codeVerifier, {
    httpOnly: true,
    secure: true,
    sameSite: 'lax',
    signed: true,
    maxAge: 60 * 1000,
  })

  res.cookie('provider', provider, {
    httpOnly: true,
    secure: true,
    sameSite: 'lax',
    signed: true,
  })

  res.redirect(authServer)
})
```

Listing 4: Login route

4. **Authorization token redirection**
   After authenticating the user, the authorization server redirects the user to the client frontend, which was defined as a *redirect URI*. The redirect includes an authorization code as well as the previously defined state parameter.

5. **Prompt for user consent**
   The client asks the user to confirm that they wish to authenticate using the chosen provider. This step is not part of the authorization code grant flow, but is included as an additional guard against CSRF. The additional protection is needed, as a user could be tricked to initiate the authorization process by clicking a malicious link. As the user is redirected to this page by the *redirect URI* parameter set in step 3, a user authenticating using a specific provider can only be redirected to one confirmation page, as long as the authorization server is not incorrectly configured. This protection is provided by the fact that authorization

servers require setting all allowed redirect URIs, and as only one URI is needed for the flow, redirection can be limited to one allowed path per provider. After consenting to the authorization, the user is redirected to the backend, forwarding the authorization code and state parameter the authorization server sent.
The consent page is implemented as the following React [46] component.

```
import { useParams } from 'react-router-dom'

const API_URL = import.meta.env.API_URL

export const OAuthCallbackPage = () => {
  const { provider } = useParams()
  const { searchParams } = new URL(document.location.
   toString())
  return (
    <>
      <h1>Login confirmation</h1>
      {provider === undefined ? (
        <p>Authentication error</p>
      ) : (
        <>
          <p>Do you wish to continue authenticating using {
   provider}?</p>
          <button
            onClick={() =>
              (window.location.href = `${API_URL}/oauth/
   code/${provider}?${searchParams.toString()}`)
            }
          >
            Continue
          </button>
        </>
      )}
    </>
  )
}
```

Listing 5: User consent page

6. **Request access token**
   The backend fetches the state, code verifier and provider that were stored as cookies in step 2. The previously stored state and provider are compared with the values in the current request, and if either value does not match the authorization process is aborted. The following steps are taken to validate the request:

```
oauthRouter.get('/code/:provider', async (req: Request, res
   : Response) => {
```

```
  // Verify that a authorization code was provided
  // and that a code verifier has been stored in the client cookies
  const code = req.query?.code
  const codeVerifier = req.signedCookies?.code_verifier
  if (typeof code !== 'string' || typeof codeVerifier !== 'string') {
    return res.sendStatus(400)
  }

  // Verify that the state sent in the original authorization request matches
  // with the state provided in the callback
  const codeState = req.query?.state
  const cookieState = req.signedCookies['state'] as string | undefined
  if (
    typeof codeState !== 'string' ||
    typeof cookieState !== 'string' ||
    codeState === '' ||
    codeState !== cookieState
  ) {
    return res.sendStatus(403)
  }

  // Verify that the provider of the oauth callback matches with the
  // provider of the original login request
  const previousProvider = req.signedCookies?.provider
  const provider = req.params.provider
  if (
    typeof previousProvider !== 'string' ||
    typeof provider !== 'string' ||
    provider !== previousProvider
  ) {
    return res.sendStatus(403)
  }
  ...
})
```

Listing 6: Part of the backend code callback route that deals with validating the request

If all the validation steps pass, an access token is requested from the authorization server based on the authorization code. It includes the same client id as in the

original authorization request, with an additional client secret. The original code verifier and redirect URI are also included.
The access token is requested with the following function:

```
export const getToken = async (
  code: string,
  codeVerifier: string,
  provider: string
): Promise<TokenResponse | undefined> => {
  const endpoint = await getEndpoint('token_endpoint', provider)
  const configCredentials = config.providers[provider]
  if (endpoint === undefined || configCredentials === undefined) {
    return undefined
  }

  const client_id = configCredentials.clientId
  const client_secret = configCredentials.clientSecret
  const redirect_uri = `${config.redirectUri}/${provider}`
  const grant_type = 'authorization_code'
  const code_verifier = codeVerifier

  try {
    const authResponse = await fetch(endpoint, {
      method: 'POST',
      body: new URLSearchParams({
        client_id,
        client_secret,
        redirect_uri,
        grant_type,
        code,
        code_verifier,
      }),
      headers: {
        'Content-Type': 'application/x-www-form-urlencoded',
      },
    })
    if (!authResponse.ok) {
      return undefined
    }
    return (await authResponse.json()) as TokenResponse
  } catch (error) {
    console.error(error)
    return undefined
  }
}
```

Listing 7: Function used to request an access token and refresh token

7. **Access token response**
   The authorization server verifies the code verifier and checks that the redirect URI matches with the URI of the original code request. If the checks pass, the server responds with an access token, as well as an optional refresh token and id token, if they were requested in the code request.

8. **Store access token**
   The backend stores the access token and optional refresh token as cookies and redirects the user to the landing page of the frontend. The token storage is described in greater detail in section 5.2. When the user makes a request to the backend in the future, the server will be able to identify the user based on this access token.
   The token storage and user redirection is handled as follows:

```
oauthRouter.get('/code/:provider', async (req: Request, res
    : Response) => {
  ...
  const token = await getToken(code, codeVerifier, provider
   )
  if (token !== undefined) {
    res.cookie('access_token', token.access_token, {
      httpOnly: true,
      secure: true,
      sameSite: 'strict',
      signed: true,
      maxAge: token.expires_in * 1000,
    })

    res.cookie('refresh_token', token.refresh_token, {
      httpOnly: true,
      secure: true,
      sameSite: 'strict',
      signed: true,
    })
  }

  res.redirect(config.frontendOrigin)
})
```

Listing 8: Part of the backend code callback route that deals with storing tokens and redirecting the user to the frontend

5.1.2 Inspecting existing access tokens

OAuth access tokens are typically opaque, and as such a client can not extract any information from the token itself. If an OAuth client wants to identify a user and confirm that an access token is valid, it must send an introspection request to the authorization server. The token introspection flow is described in figure 7. The token introspection relies on an access token and a provider that were stored when obtaining an access token. Many frontend libraries require explicitly including cookies to send them in any requests. In our TypeScript example using the fetch() global function [16], cookies can be included with the *credentials: 'include'* option. The backend application must also set the Access-Control-Allow-Credentials header in the response to signal that cookies are used to identify requests.

Based on the type of token, the authorization server can return a differing amount of information, such as user emails and names. The minimal information that it will return however is a unique *sub* value, which is a unique identifier for the token subject. When combining this sub value with the token provider, the OAuth client can identify a user, allowing the client application to serve personalized content.

5.1.3 Refreshing access tokens

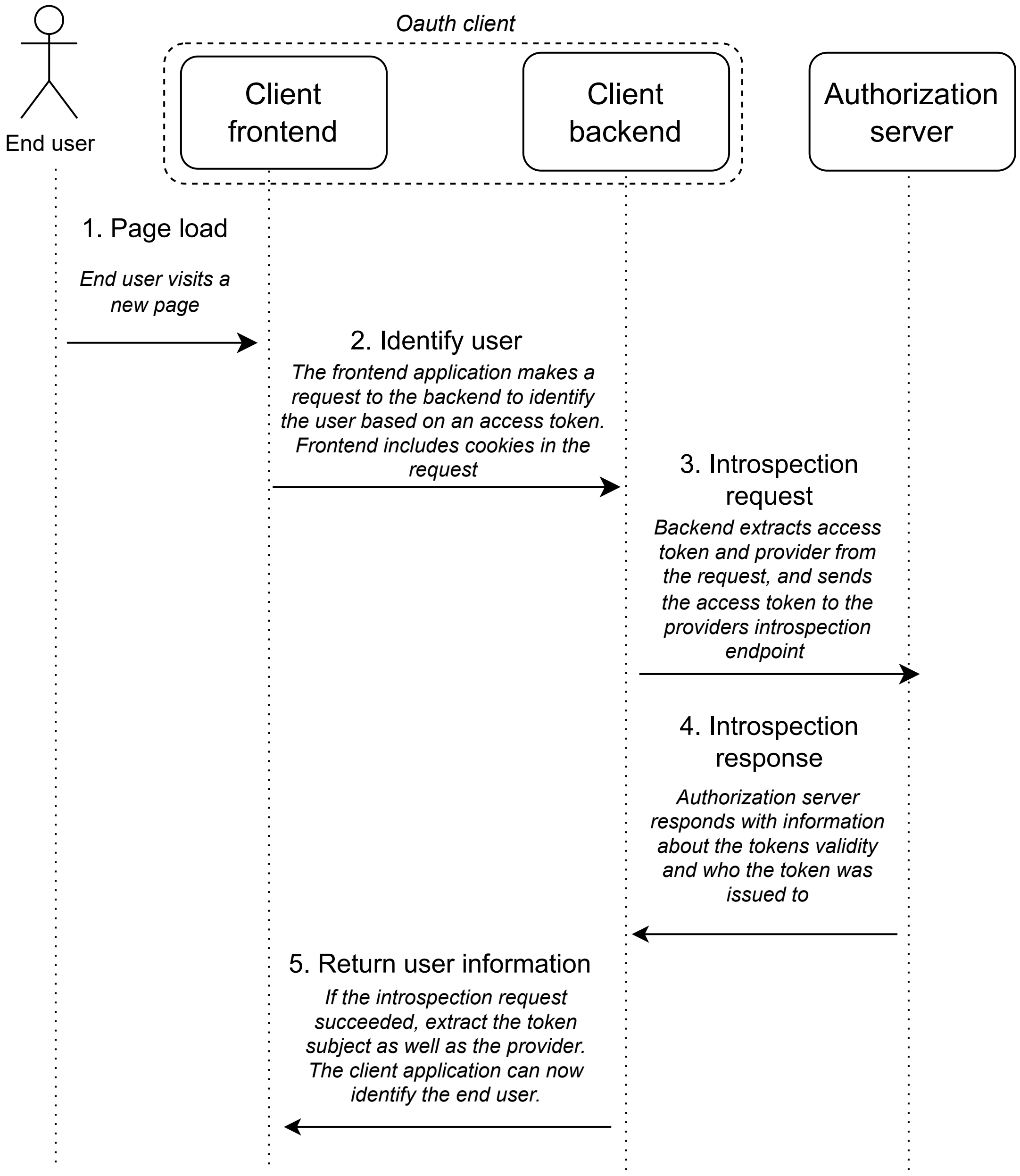
Since access tokens are only valid for a specific time, it is useful to be able to refresh existing tokens, ideally without requiring user input. To achieve this, OAuth clients can request refresh tokens, which can be used to request new access tokens. While they do improve the user experience, refresh tokens weaken the security of the system, as their long expiry time increases the probability that they will be leaked.

In the client example, a refresh token is requested by setting the $access\_type$ to offline in the authorization code request (step 3 in figure 6). The refresh token is then stored as a cookie in the end user's browser, similarly to the access token. The refresh token can now be used when the client frontend interacts with the backend. In the client example, the refresh token is used in combination with the token introspection request. If a refresh token is set and the access token is not set, or if the introspection response fails (step 4 in figure 7), the backend server will attempt to request a new access token based on the stored refresh token. If the refresh succeeds, the client stores the updated access token and retries the earlier token introspection. If the refresh fails, the refresh token and access token are removed.

5.2 Token storage

As described in section 2.3, a number of options exist for storing data in web browsers. The most secure storage solution for an OAuth client would be to temporarily store tokens in JavaScript, with closures providing some additional security. Such a solution would not however provide any persistence, and as such the end user would have to authenticate every time they start a new browser session. To improve the user experience, the example client persists the access tokens, allowing users to stay logged in across multiple sessions. When choosing to persist data developers have two practical options: the web storage API and cookies. Due to its vulnerability to XSS attacks, the web storage API is not an optimal solution for storing sensitive information. We instead opt for cookies to store any client-specific information when obtaining an access token, described in section 5.1.1.

Many cookie flags that improve the storage security, described in section 2.3, are essential when storing sensitive data such as access tokens. The cookies are marked as *HttpOnly*, causing the cookies to be included in requests while keeping them completely inaccessible for the frontend application. Instead, only the backend application can set and modify the cookies. Any sensitive cookies should also be



Oauth client
End user
Client frontend
Client backend
Authorization server
1. Page load
End user visits a new page
2. Identify user
The frontend application makes a request to the backend to identify the user based on an access token. Frontend includes cookies in the request
3. Introspection request
Backend extracts access token and provider from the request, and sends the access token to the providers introspection endpoint
4. Introspection response
Authorization server responds with information about the tokens validity and who the token was issued to
5. Return user information
If the introspection request succeeded, extract the token subject as well as the provider. The client application can now identify the end user.

Figure 7: Network traffic of the access token introspection

marked as *Secure*, causing them to only be included in HTTPS requests. This is a critical flag to include, as any network attacker could intercept sensitive cookies in transit if they are included in regular HTTP traffic. Cookies should also have an appropriate *SameSite* value. The state, code verifier and provider should be stored with the *Lax* SameSite policy, as they are accessed in a request that is initiated from the authorization server. The access token and refresh token should instead be stored with the *Strict* policy, as it does not need to be included in requests with any external origin.

To further limit the possibility of sensitive tokens leaking, tokens can be encrypted before storage. Since all communication with the authorization server in an OAuth client with a separate frontend and backend flows through the backend server, the end user does not need access to the original, unencrypted access token, code verifier or OAuth provider. If all the values stored in the client are encrypted, an attacker is unable to steal the original tokens. The provided security benefit from this is however quite limited, as the client backend will accept and decrypt any token, without any way to verify if it was stolen or not. Instead, the encrypted cookies have a greater impact on the authorization code flow, as modifying the temporarily stored state, code verifier or OAuth provider is more difficult. An attacker is required to interact with the backend client themselves to obtain modified values that are encrypted with the correct key, allowing the client to limit what type of values are generated. This also makes it easier to investigate malicious activity, since the attacker has to interact with the legitimate client to pull off any attack.

5.3 Proof Key for Code Exchange

To protect against authorization code injection, the OAuth 2.0 Security Best Current Practice [47] recommends using Proof Key for Code Exchange (PKCE) to bind authorization codes to client instances [48].

PKCE relies on creating a code verifier in the initial authorization code request (step 2 in figure 6). The code verifier should be a random string with between 43 and 128 characters. This code verifier should be temporarily stored by the client. In our example client, the value is stored as a cookie in the user's browser. After generating the code verifier, the client generates a code challenge by hashing the code verifier. The code challenge should be generated using the SHA256 algorithm [49] if possible. The generated code challenge is sent to the authorization server together with the code request. The used code challenge method, such as *S256* if SHA256 is used, is also included in the request.

After receiving the authorization code, the OAuth client requests an access token from the authorization server (step 6 in figure 6). The original code verifier is included in the access token request. The authorization server computes a code challenge based on the verifier and the previously sent challenge method and compares it with the initial code challenge. If the two code challenge values do not match, the authorization server interrupts the access token generation.

5.4 Explicit user intention tracking

To protect against CSRF attacks and IdP mix-up attacks, described in sections 4.2.1 and 4.2.2, additional controls should be used to keep track of user actions. To alleviate these issues, the authorization code flow described in figure 6 contains some additional checks that are not mandatory in the OAuth 2.0 authorization code flow described in figure 4. The chosen provider is tracked in each step, with separate routes for separate providers to limit the chance for a mix-up. Additionally, the original chosen provider is stored as a cookie, allowing the client to validate that the actual used provider is the same as the provider chosen by the user.

Step 5 of figure 6 is also added as an extra guard against CSRF and IdP mix-up attacks. The redirect via the frontend prompting for user consent is not required to request an access token, which could be handled completely in the backend. It is however useful as a prompt for the user to confirm that they intend to sign in to the page, and that they chose to sign in with a specific provider. If the step was missing, it is possible that an attacker could craft a CSRF request that signs in the user as the attacker, or that signs in the user using the wrong authorization server.

6 Discussion

This section discusses the findings of sections 4 and 5, with discussion about the chosen implementation decisions and other possible options as well as their benefits and drawbacks. Observations from utilizing third-party authorization servers are also detailed, with a focus on how securely the servers are implemented, and how well they succeed in guiding developers to implement secure clients through their documentation.

6.1 Client implementation options

When implementing an OAuth client, the security of two clients that both adhere to the OAuth 2.0 specification can be very different. This is mainly due to the specification being quite lenient, with a significant portion of security-enhancing features defined as optional or recommended. While this does have the benefit of allowing developers to create very simple OAuth clients that still adhere to the specification, the varying security can be misleading for end users that are likely to assume that all single sign-on systems are created equal.

6.1.1 OAuth authorization flows

A general design principle for our OAuth client is to do as much in the client backend as possible, as opposed to in the frontend. While the frontend is a SPA and could thus do almost everything the backend does, every frontend operation has to consider the possibility that it could be tampered with by an attacker. Keeping the frontend implementation as simple as possible also makes it easier to translate the logic to other languages or frameworks if required. As described in section 5, the separate client backend also allows keeping values secret from the end user, which is required to implement the authorization code flow, which should be wherever possible instead of the less secure implicit grant.

The authorization code flow requires the client to generate random values for the state and code verifier. Generating random values is a very common problem with a number of viable solutions. Our client uses the *randomBytes* function from the *crypto* module to generate these random values [50]. The *crypto* module has the benefit of being included in Node.Js by default, without requiring any additional third-party libraries. The widespread use of the library also makes it more likely that any vulnerabilities are found and patched promptly. The code challenge could also be sent in plain text, so that the code challenge and code verifier are the same string. If possible, the code challenge should however be a hashed version of the code verifier. The Node.Js *crypto* module contains a function for generating the hash as well, *createHash*, which is used in our client, using the SHA256 algorithm. While authorization servers can choose to support other hashing algorithms, they are only required to support SHA256 and plain text code challenges [48].

Both the frontend and backend keep track of the used authorization server by using separate paths for separate providers. The provider could be provided as a query parameter as well, with no impact on the security of the system. It is however simpler to keep track of the provider in the path, as the authorization server redirect contains data as query parameters, which could potentially cause clashes with any additional data stored using the same parameters. Other options are not available, such as sending the provider in the request body, as the provider has to be contained in the redirect URI sent to the authorization server. By additionally storing the provider as a cookie, the backend server can confirm that the initial login request was using the provider that the user believes they are using when approving the login on the consent page. This is necessary, as a user could be redirected to the confirmation page from any server with any parameters, legitimate or malicious, with no way to confirm where the redirect request originated.

The additional user consent screen is mainly intended as a defense mechanism against IdP mix-up attacks and CSRF attacks. CSRF attacks generally rely on users clicking a link which has unintended side effects, such as causing the user to submit a form with the values as query parameters in the link. While the client frontend does not have a form that can be submitted to initiate the login request, a link to the backend server with the correct URI would cause the user to be redirected to an authorization server. As such, it has to be considered that the user could be redirected to the redirect URI defined in the authorization code request without ever knowingly agreeing to sign in. Since the allowed redirect URIs have to be defined for each authorization server, if each provider has separate callback paths, a sign-in request from a legitimate authorization server can only redirect to their own callback path. While it is still possible for an attacker to redirect a user to this confirmation page with malicious parameters, such as a code belonging to the attacker, the backend server should be able to catch these attacks by verifying the state and code verifier values. The state and code verifier values created in the initial code request are only stored in the client's browser, and due to them being single-origin, the backend server must have set the cookies if they are able to read them. If an attacker crafts a malicious link to request an access token based on the attacker's code, both the state and the code verifier checks will fail. Additionally, since authorization servers should perform exact URI comparisons to verify that a redirect URI is allowed, an attacker is unable to embed any malicious content in the redirect URI itself.

6.1.2 Persisting OAuth tokens

While the improvements in user experience are clear when persisting access tokens using some client-side storage method, such as cookies in our example, long-time storage of tokens does make the system less secure. While cookies marked as *HttpOnly* are not accessible from applications running in the client browser, the end user is able to access them in the browser. Any token that is stored runs the risk of falling into the wrong hands, which is why many critical applications, such as online banks, often do not allow users to stay logged in between multiple sessions. Access tokens do mitigate this risk of leaking somewhat by being relatively short lived.

Refresh tokens are however not short lived, and as described in section 6.2.2, authorization servers often allow refresh tokens to be used to request multiple access tokens, without having an expiration time for the refresh token. As such, refresh tokens make the system less secure. Furthermore, if refresh tokens are stored in the same place as access tokens, as is done in our OAuth client example, the benefits of having an expiration time on the access tokens are not realized, as it is equally likely for the refresh token and access token to be leaked, while the refresh token can be valid for a longer time increasing the chance that an attacker is able to exploit it. This implies that refresh tokens should not be stored in cookies together with the access token, and in our client example this could be achieved by storing the refresh token in the client backend, which should limit the chance for these tokens to be leaked. This immediately raises the requirements for developers and system administrators, as they are now responsible for implementing a secure storage method for these sensitive tokens. This mitigates one of the main benefits of utilizing third party authorization servers, which is the lowered requirements for a developer wishing to implement a secure authorization system.

6.2 Observations from utilizing third-party authorization servers

A clear benefit of the OAuth 2.0 protocol is that it removes many complicated technical details from application developers, allowing them to rely on an established and trusted third party to implement the critical sign-in logic, leaving less room for mistakes. For this benefit to materialize, the application developers should be able to find information on how clients can be implemented securely. Additionally, as the third parties are trusted to implement the sign-in system securely, their authorization servers should be implemented correctly and securely. This section describes the developer experience of using authorization servers provided by Google and Microsoft, with notes on how securely they are implemented.

6.2.1 Quality of documentation

Since developers using third party authorization servers might not be familiar with OAuth best practices, it is essential that documentation and examples show secure implementations. As discussed in chapters 4 and 5, a developer implementing an OAuth client has to make a number of decisions that have a significant impact on the security of the system. Some of the areas where developers can make decisions that impact the system security are the choice of authorization flow, the use of state and PKCE and the use of refresh tokens.

As discussed in sections 3.3.1 and 3.3.2, the authorization code flow should be preferred over the implicit grant flow whenever possible due to the greatly improved security. Despite this, the Google documentation for client-side web applications state that the implicit grant flow is the only viable option for such clients [51]. In contrast, Microsoft's version of the same documentation clearly states that the implicit grant flow is not a suitable authentication method today, and even states that existing applications using the implicit grant flow should migrate to using the authorization code flow[52].

When utilizing the authorization code flow, using the state value is a simple way to protect against a number of attacks, as discussed in section 4.2.1. As such, it is a positive sign that both studied documentation sets recommend using the state value, and that the state value is included in example requests. While the inclusion of the state value is positive, neither of the used documentation sets actually explain how or where the state value should be verified, leaving some room for improvement.

Similarly to the recommended authorization flow, Google and Microsoft seem to have a very different view on the importance of PKCE. In the case of Google, PKCE is only described in their documentation for mobile and desktop applications [53], while it is completely absent from their documentation for web servers [54]. Even in the desktop documentation where PKCE is described, it is completely missing from the example requests, leaving it up to the developer to figure out how and where the *code_challenge* and *code_verifier* values should be included. Microsoft's documentation on the other hand does describe PKCE, and the relevant parameters are included in the example requests. Additionally, the Microsoft documentation recommends using PKCE for all application types and requires its use for single page applications that use the authorization code flow.

6.2.2 Authorization server security

When choosing to use third-party authorization servers, developers rely on these third parties to securely implement the services. Since the OAuth 2.0 specification is quite lax, an authorization server that complies with the specification can have a number of known vulnerabilities. Details on how authorization servers are implemented are provided below, based on observations from utilizing Microsoft's and Google's authorization servers and the vulnerabilities listed in section 4.1.

Supporting the use of state is essential when implementing secure OAuth clients. As such, it is a positive note that the investigated authorization servers support the state parameter, and have implemented it correctly. The state parameter is however optional, leaving room for developers to implement insecure clients. While the state parameter is described as optional in the OAuth specification, authorization servers could encourage more secure practices by requiring users to provide a state value. The state support does however leave room for improvement, as the servers do not support invalidating a code if a state validation fails. While this code invalidation is not described in the OAuth specification, it does provide clear security benefits, as a failure to validate the state should only occur as a result of an error or some malicious activity. As the authorization codes are single-use, this invalidation could be implemented by requesting and immediately invalidating an access token, but a more direct option would be useful.

Due to the distributed nature of the internet, sensitive traffic should never be sent without encryption. Since OAuth deals with sign-in logic and user credentials, there is no reason why any of the communication should use plain HTTP. Microsoft and Google seem to agree, since all their endpoints force clients to use HTTPS. Microsoft also forces redirect URIs to use HTTPS, only allowing for localhost addresses to use plain HTTP. Google does however allow using HTTP for non-local redirect URIs.

Since authorization codes and access tokens are sent to the redirect URI, any network attacker that is able to intercept the traffic would be able to read this sensitive data in plain-text.

Section 4.1 also discusses a number of other redirect-related vulnerabilities. Both of the studied authorization servers correctly mitigate these vulnerabilities. The HTTP 307 redirect vulnerability relies on authorization servers incorrectly redirecting the request body, potentially allowing clients to receive user credentials. While both studied authorization servers use HTTP 302 instead of the 303 code recommended by Fett et al. (2016) [2], the body is correctly removed from the redirect response. The redirect URI attack relies on authorization servers not performing exact redirect URI matching when comparing provided redirect URIs with the list of allowed URIs, such as only comparing the origin. Both studied authorization servers do perform exact URI matching, and as such they are not vulnerable to the attack.

Insecure handling of tokens is another common source for vulnerabilities in authorization servers. The studied authorization servers correctly implement authorization codes by only allowing them to be used once. The validity times for the access tokens are also appropriate, with both the access tokens of both providers being valid for one hour. Both providers do however implement refresh tokens in a less secure manner. Ideally, refresh tokens should have a similar expiration time as access tokens, requiring clients to periodically renew the token to keep it valid. Additionally, refresh tokens should be single use, with authorization servers returning the next refresh token in the access token refresh response. Instead, both providers issue refresh tokens that do not expire. Additionally, the same refresh token can be used to request multiple access tokens. Due to these implementation choices, the risk for session hijacking is increased significantly due to old leaked refresh tokens still being valid. On a positive note, the authorization servers do correctly invalidate refresh tokens when the related access token is invalidated.

7 Conclusions

This thesis studied common vulnerabilities of systems using OAuth 2.0 and presented methods that can be used to mitigate these issues. The motivation behind the study was the widespread use of OAuth and the lax nature of the standard, allowing clients to be implemented with very different levels of security. The goal of the thesis was to identify common vulnerabilities in OAuth systems and demonstrate how a client can be implemented securely.

A number of common vulnerabilities in OAuth clients and authorization servers were described, in addition to steps that can be taken to mitigate the vulnerabilities. The studied vulnerabilities can be divided into two categories: vulnerabilities caused by incorrect authorization server or client implementations and vulnerabilities in OAuth systems that implement the bare minimum required by the specification. Incorrect handling of the authorization code redirect is shown to be a common source for vulnerabilities, such as the redirect URI attack which is made possible by systems not implementing exact URI matching for the redirect URI and the HTTP 307 attack, which is enabled by authorization servers sending the original request body in the redirect response. Vulnerabilities that are caused by systems implementing the bare minimum of the OAuth specification are most commonly related to not using the state parameter and authorization servers allowing unencrypted web traffic, either to their own authorization endpoint or when redirecting back to the client.

The practical client implementation of the study shows that implementing a secure OAuth client is simple, as long as a few crucial details are implemented correctly. While the client is implemented using certain tools and frameworks, the implementation is simple enough that it should be possible to translate the logic to any language or framework of choice. The client implementation also showed that developers can outsource a significant amount of complexity related to user authentication to third parties that provide authorization servers. While the steps needed to implement a secure OAuth client are not complex, the lax nature of the OAuth specification makes it easy for developers to make insecure decisions if they are not familiar with the technologies used.

Many of the mitigations were shown to be simple and inexpensive to implement, with the OAuth specification describing a significant portion of the security enhancing techniques. Techniques that are outside of the OAuth 2.0 specification, such as PKCE, are similarly shown to be simple and effective at preventing many vulnerabilities. Due to how easy many of these mitigations are to implement, it is not obvious why they are not mandatory. While a more secure specification will still have to be implemented correctly by developers, explicitly requiring more secure processes should help limit the number of vulnerabilities found in real world applications.

References

[1] R. Dhamija and L. Dusseault. "The Seven Flaws of Identity Management: Usability and Security Challenges". In: *IEEE Security & Privacy Magazine* 6.2 (Mar. 2008), pp. 24–29. ISSN: 1540-7993. DOI: 10.1109/MSP.2008.49.

[2] D. Fett, R. Küsters, and G. Schmitz. "A Comprehensive Formal Security Analysis of OAuth 2.0". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. New York, NY, USA: Association for Computing Machinery, Oct. 2016, pp. 1204–1215. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978385.

[3] M. E. Hellman. "An Overview of Public Key Cryptography". In: *IEEE COMMUNICATIONS SOCIETY MAGAZINE* (1978).

[4] W. Diffie and M. Hellman. "New directions in cryptography". In: *IEEE Transactions on Information Theory* 22.6 (Nov. 1976), pp. 644–654. ISSN: 0018-9448, 1557-9654. DOI: 10.1109/TIT.1976.1055638.

[5] D. Gillmor. *Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS)*. Tech. rep. RFC7919. RFC Editor, Aug. 2016, RFC7919. DOI: 10.17487/RFC7919.

[6] G. Avoine, S. Canard, and L. Ferreira. "Symmetric-Key Authenticated Key Exchange (SAKE) with Perfect Forward Secrecy". In: *Topics in Cryptology – CT-RSA 2020*. Ed. by S. Jarecki. Vol. 12006. Cham: Springer International Publishing, 2020, pp. 199–224. ISBN: 978-3-030-40185-6 978-3-030-40186-3. DOI: 10.1007/978-3-030-40186-3_10.

[7] B. Dowling et al. "A Cryptographic Analysis of the TLS 1.3 Handshake Protocol Candidates". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. Denver Colorado USA: ACM, Oct. 2015, pp. 1197–1210. ISBN: 978-1-4503-3832-5. DOI: 10.1145/2810103.2813653.

[8] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. Tech. rep. RFC 8446. Internet Engineering Task Force, Aug. 2018. DOI: 10.17487/RFC8446.

[9] K. Jacksi and S. M. Abass. "Development History Of The World Wide Web". In: *International Journal of Scientific & Technology Research* 8.09 (2019). ISSN: 2277-8616.

[10] R. Camden and B. Rinaldi. "Why Static Sites?" In: *Working with Static Sites*. 2017, pp. 1–8. ISBN: 978-1-4919-6094-3.

[11] K. Okoye, H. Jahankhani, and A.-R. H. Tawil. "Accessibility of dynamic web applications with emphasis on visually impaired users". In: *The Journal of Engineering* 2014.9 (2014), pp. 531–537. ISSN: 2051-3305. DOI: 10.1049/joe.2014.0136.

[12] K. Nath, S. Dhar, and S. Basishtha. "Web 1.0 to Web 3.0 - Evolution of the Web and its various challenges". In: *2014 International Conference on Reliability Optimization and Information Technology (ICROIT)*. Feb. 2014, pp. 86–89. DOI: 10.1109/ICROIT.2014.6798297.

[13] P. Fraternali, G. Rossi, and F. Sánchez-Figueroa. "Rich Internet Applications". In: *IEEE Internet Computing* 14.3 (May 2010), pp. 9–12. ISSN: 1941-0131. DOI: 10.1109/MIC.2010.76.

[14] *Adobe Flash Player : Security vulnerabilities, CVEs*. URL: https://www.cvedetails.com/vulnerability-list/vendor_id-53/product_id-6761/Adobe-Flash-Player.html (visited on 01/31/2024).

[15] *jQuery*. URL: https://jquery.com/ (visited on 01/31/2024).

[16] *fetch() global function - Web APIs | MDN*. Nov. 2023. URL: https://developer.mozilla.org/en-US/docs/Web/API/fetch (visited on 01/31/2024).

[17] G. Fink and I. Flatow. "Introducing Single Page Applications". In: *Pro Single Page Application Development: Using Backbone.js and ASP.NET*. Berkeley, CA: Apress, 2014, pp. 3–13. ISBN: 978-1-4302-6674-7. DOI: 10.1007/978-1-4302-6674-7_1.

[18] *HTML Standard - Web storage*. URL: https://html.spec.whatwg.org/multipage/webstorage.html (visited on 02/13/2024).

[19] A. Barth. *HTTP State Management Mechanism*. Tech. rep. RFC 6265. Apr. 2011. DOI: 10.17487/RFC6265.

[20] S. Khodayari and G. Pellegrino. "The State of the SameSite: Studying the Usage, Effectiveness, and Adequacy of SameSite Cookies". In: *2022 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, May 2022, pp. 1590–1607. ISBN: 978-1-66541-316-9. DOI: 10.1109/SP46214.2022.9833637.

[21] *Web Storage API - Web APIs | MDN*. Dec. 2023. URL: https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API (visited on 02/13/2024).

[22] J. Reschke. *The 'Basic' HTTP Authentication Scheme*. Request for Comments RFC 7617. Internet Engineering Task Force, Sept. 2015. DOI: 10.17487/RFC7617.

[23] A. Das et al. "The Tangled Web of Password Reuse". In: *Proceedings 2014 Network and Distributed System Security Symposium*. Internet Society, 2014. ISBN: 978-1-891562-35-8. DOI: 10.14722/ndss.2014.23357.

[24] S. Farrell. "API Keys to the Kingdom". In: *IEEE Internet Computing* 13.5 (Sept. 2009), pp. 91–93. ISSN: 1941-0131. DOI: 10.1109/MIC.2009.100.

[25] D. Hardt. *The OAuth 2.0 Authorization Framework*. Request for Comments RFC 6749. Internet Engineering Task Force, Oct. 2012. DOI: 10.17487/RFC6749. URL: https://datatracker.ietf.org/doc/rfc6749.

[26] X. Lin et al. “Threat Modeling for CSRF Attacks”. In: *2009 International Conference on Computational Science and Engineering*. Vancouver, BC, Canada: IEEE, 2009, pp. 486–491. ISBN: 978-1-4244-5334-4. DOI: 10.1109/CSE.2009.372.

[27] A. Barth, C. Jackson, and J. C. Mitchell. “Robust defenses for cross-site request forgery”. In: *Proceedings of the 15th ACM conference on Computer and communications security*. Alexandria Virginia USA: ACM, Oct. 2008, pp. 75–88. ISBN: 978-1-59593-810-7. DOI: 10.1145/1455770.1455782.

[28] H. Kim and E. A. Lee. “Authentication and Authorization for the Internet of Things”. In: *IT Professional* 19.5 (2017), pp. 27–33. ISSN: 1520-9202. DOI: 10.1109/MITP.2017.3680960.

[29] E. Hammer-Lahav. *The OAuth 1.0 Protocol*. Request for Comments RFC 5849. Internet Engineering Task Force, Apr. 2010. DOI: 10.17487/RFC5849.

[30] *POST - HTTP | MDN*. Dec. 2023. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST (visited on 02/08/2024).

[31] J. Richer. *OAuth 2.0 Token Introspection*. Request for Comments RFC 7662. Internet Engineering Task Force, Oct. 2015. DOI: 10.17487/RFC7662.

[32] P. Philippaerts, D. Preuveneers, and W. Joosen. “OAuch: Exploring Security Compliance in the OAuth 2.0 Ecosystem”. In: *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*. RAID ’22. New York, NY, USA: Association for Computing Machinery, Oct. 2022, pp. 460–481. ISBN: 978-1-4503-9704-9. DOI: 10.1145/3545948.3545955.

[33] R. Yang et al. “Model-based Security Testing: An Empirical Study on OAuth 2.0 Implementations”. In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ASIA CCS ’16. New York, NY, USA: Association for Computing Machinery, May 2016, pp. 651–662. ISBN: 978-1-4503-4233-9. DOI: 10.1145/2897845.2897874.

[34] R. Dhamija, J. D. Tygar, and M. Hearst. “Why phishing works”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Montréal Québec Canada: ACM, Apr. 2006, pp. 581–590. ISBN: 978-1-59593-372-0. DOI: 10.1145/1124772.1124861.

[35] R. T. Fielding, M. Nottingham, and J. Reschke. *HTTP Semantics*. Request for Comments RFC 9110. Internet Engineering Task Force, June 2022. DOI: 10.17487/RFC9110.

[36] F. Yang and S. Manoharan. “A security analysis of the OAuth protocol”. In: *2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*. ISSN: 2154-5952. Aug. 2013, pp. 271–276. DOI: 10.1109/PACRIM.2013.6625487.

[37] M. Darwish and A. Ouda. “Evaluation of an OAuth 2.0 protocol implementation for web server applications”. In: Oct. 2015, pp. 1–4. DOI: 10.1109/IEMCON.2015.7344461.

[38] W. Li, C. J. Mitchell, and T. Chen. “Your Code Is My Code: Exploiting a Common Weakness in OAuth 2.0 Implementations”. In: *Security Protocols XXVI*. Ed. by V. Matyáš et al. Vol. 11286. Cham: Springer International Publishing, 2018, pp. 24–41. ISBN: 978-3-030-03250-0 978-3-030-03251-7. DOI: 10.1007/978-3-030-03251-7_3.

[39] E. Shernan et al. “More Guidelines Than Rules: CSRF Vulnerabilities from Noncompliant OAuth 2.0 Implementations”. In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by M. Almgren, V. Gulisano, and F. Maggi. Vol. 9148. Cham: Springer International Publishing, 2015, pp. 239–260. ISBN: 978-3-319-20549-6 978-3-319-20550-2. DOI: 10.1007/978-3-319-20550-2_13.

[40] W. Li and C. J. Mitchell. “Security Issues in OAuth 2.0 SSO Implementations”. In: *Information Security*. Ed. by S. S. M. Chow et al. Vol. 8783. Cham: Springer International Publishing, 2014, pp. 529–541. ISBN: 978-3-319-13256-3 978-3-319-13257-0. DOI: 10.1007/978-3-319-13257-0_34.

[41] M. Argyriou, N. Dragoni, and A. Spognardi. “Security Flows in OAuth 2.0 Framework: A Case Study”. In: *Computer Safety, Reliability, and Security*. Ed. by S. Tonetta, E. Schoitsch, and F. Bitsch. Vol. 10489. Cham: Springer International Publishing, 2017, pp. 396–406. ISBN: 978-3-319-66283-1 978-3-319-66284-8. DOI: 10.1007/978-3-319-66284-8_33.

[42] A. Taly et al. “Automated Analysis of Security-Critical JavaScript APIs”. In: *2011 IEEE Symposium on Security and Privacy*. Oakland, CA, USA: IEEE, May 2011, pp. 363–378. ISBN: 978-1-4577-0147-4. DOI: 10.1109/SP.2011.39.

[43] P. Chinprutthiwong et al. “Security Study of Service Worker Cross-Site Scripting.” In: *Annual Computer Security Applications Conference*. Austin USA: ACM, Dec. 2020, pp. 643–654. ISBN: 978-1-4503-8858-0. DOI: 10.1145/3427228.3427290.

[44] *backjonas/oauth-client*. URL: https://github.com/backjonas/oauth-client (visited on 05/27/2024).

[45] *Express routing*. URL: https://expressjs.com/en/guide/routing.html (visited on 05/20/2024).

[46] *React*. URL: https://react.dev/ (visited on 05/20/2024).

[47] T. Lodderstedt et al. *OAuth 2.0 Security Best Current Practice*. Internet-Draft draft-ietf-oauth-security-topics-27. Work in Progress. Internet Engineering Task Force, May 2024. 59 pp. URL: https://datatracker.ietf.org/doc/draft-ietf-oauth-security-topics/27/.

[48] N. Sakimura, J. Bradley, and N. Agarwal. *Proof Key for Code Exchange by OAuth Public Clients*. RFC 7636. Sept. 2015. DOI: 10.17487/RFC7636.

[49] Q. H. Dang. *Secure Hash Standard*. Tech. rep. NIST FIPS 180-4. National Institute of Standards and Technology, July 2015, NIST FIPS 180–4. DOI: 10.6028/NIST.FIPS.180-4.

[50] *Crypto | Node.js v22.2.0 Documentation*. URL: https://nodejs.org/api/crypto.html (visited on 05/20/2024).

[51] *OAuth 2.0 for Client-side Web Applications | Authorization | Google for Developers*. URL: https://developers.google.com/identity/protocols/oauth2/javascript-implicit-flow (visited on 05/14/2024).

[52] OwenRichards1. *Microsoft identity platform and OAuth 2.0 implicit grant flow - Microsoft identity platform*. Apr. 2024. URL: https://learn.microsoft.com/en-us/entra/identity-platform/v2-oauth2-implicit-grant-flow (visited on 05/14/2024).

[53] *OAuth 2.0 for Mobile & Desktop Apps | Authorization*. URL: https://developers.google.com/identity/protocols/oauth2/native-app (visited on 05/14/2024).

[54] *Using OAuth 2.0 for Web Server Applications | Authorization*. URL: https://developers.google.com/identity/protocols/oauth2/web-server (visited on 05/14/2024).