ENHANCING REST API ACCESS CONTROL USING MULTIPLE FACTOR AUTHENTICATION

WITH REFRESH TOKEN

by

Ahadu B. Melesse

APPROVED BY SUPERVISORY COMMITTEE

__________________________________
Lawrence Chung, Chair

__________________________________
Ravi Prakash

__________________________________
Gity Karami
TO B.
ENHANCING REST API ACCESS CONTROL USING MULTIPLE FACTOR AUTHENTICATION

WITH REFRESH TOKEN

by

AHADU B. MELESSE, MS

THESIS

Presented to the Faculty of

The University of Texas at Dallas

in partial fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN

SOFTWARE ENGINEERING

THE UNIVERSITY OF TEXAS AT DALLAS

December 2023
ACKNOWLEDGEMENTS

All credit to the almighty, who makes it all possible. Grateful to the family; thanks for the love, words of kindness, care, and helping out in my time of needs.

I would like to thank my supervisor, Dr. Chung, the committee members (Dr. Ravi and Dr. Gity), and the team who guided me throughout the research. I would also like to express my regards to all my UTD professors.

November 2023
Representational State Transfer Application Programming Interfaces (RESTful APIs) have emerged as a crucial component in modern web applications, facilitating efficient data exchange between clients and servers and to request data process as well. Securing these APIs is paramount, with a focus on safeguarding endpoints (REST APIs). Despite the impossibility of achieving complete system security, addressing vulnerabilities, particularly in the context of public clients and exchange of access-token and refresh-token, is significant. This thesis explores the utilization of Identity Providers protocol, such as OAuth 2.0 and OpenID Connect, to bolster access control through standardized authentication and authorization methods and guidelines.

In the realm of API-driven applications protecting resources and data is not optional. The need for robust security measures has intensified. A notable challenge in RESTful API security lies in authentication and authorization, with refresh-tokens serving as a common means for clients to acquire new access-tokens without user credentials. However, the extended lifespan of refresh-tokens poses a potential security risk if compromised. This research proposes the integration of
multi-factor authentication during the use of refresh-tokens, enhancing account security and mitigating the risk of unauthorized access. This approach offers an additional layer of security without the need to revoke access or refresh-tokens. Given the dynamic nature of access management, staying abreast of the latest developments and best practices is crucial for maintaining application security. This thesis provides a concise overview of key considerations and strategies, leveraging OAuth 2.0 and OpenID Connect, along with a refresh-token-based approach using the Duende.Identity Server as a central Identity Provider. By adopting this approach, developers and organizations can fortify the security of their API-driven applications in the face of an ever-evolving threat landscape. The use of Duende.Identity Server, acknowledged even by Microsoft, ensures the implementation of security measures based on a proven protocol that addresses the concern.
# TABLE OF CONTENTS

ACKNOWLEDGEMENTS.................................................................................................................... v

ABSTRACT........................................................................................................................................ vi

LIST OF FIGURES.............................................................................................................................. xi

CHAPTER 1  INTRODUCTION .......................................................................................................... 1

1.1 Problem Statement....................................................................................................... 1
1.2 History........................................................................................................................... 2
1.3 Definition ...................................................................................................................... 3
  1.3.1 REST................................................................................................................... 3
  1.3.2 Representational State Transfer ....................................................................... 3
  1.3.3 Stateless............................................................................................................ 4
  1.3.4 API ..................................................................................................................... 6
  1.3.5 Client ................................................................................................................. 6
  1.3.6 Request ............................................................................................................. 6
  1.3.7 Response ........................................................................................................... 7
  1.3.8 Resource............................................................................................................ 7
  1.3.9 REST Server ....................................................................................................... 7
1.4 Applications Architecture ............................................................................................ 8
1.5 Storyline ..................................................................................................................... 10
  1.5.1 Connectify’s Case ............................................................................................ 10
  1.5.2 Big Firms .......................................................................................................... 10
1.6 Authentication and Authorization ............................................................................... 11
1.7 Token.......................................................................................................................... 11
  1.7.1 ID Token ......................................................................................................... 12
  1.7.2 Access Token .................................................................................................. 13
  1.7.3 Refresh Token ................................................................................................ 14
1.8 Authentication and Authorization Protocols ........................................................... 14
1.9 Authentication Service .............................................................................................. 15
1.10 Authentication-Authorization Flow ......................................................................... 15
1.11 OpenID Connect vs. OAuth 2.0............................................................................... 17
1.12 Authorization Code Flow with PKCE................................................................. 20
1.13 Code Verifier............................................................................................................ 21

CHAPTER 2  RELATED WORKS ...................................................................................................... 25

2.1 Cross Site Scripting Scenario ................................................................. 26
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>State</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Stateless</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>Request (Stateless)</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>REST API Architecture</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>API request-response</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>Modern architecture</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>Authentication-Authorization</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>ID-Token</td>
<td>13</td>
</tr>
<tr>
<td>9</td>
<td>Access-Token</td>
<td>14</td>
</tr>
<tr>
<td>10</td>
<td>Authentication-Authorization (sequence)</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>Authentication-Authorization (refresh-token)</td>
<td>17</td>
</tr>
<tr>
<td>12</td>
<td>Authentication-Authorization (workflow)</td>
<td>19</td>
</tr>
<tr>
<td>13</td>
<td>Authorization code flow</td>
<td>21</td>
</tr>
<tr>
<td>14</td>
<td>Handing authorization token</td>
<td>23</td>
</tr>
<tr>
<td>15</td>
<td>Authorization code flow with PKCE</td>
<td>24</td>
</tr>
<tr>
<td>16</td>
<td>Hacking Race</td>
<td>26</td>
</tr>
<tr>
<td>17</td>
<td>Refresh-Token rotation</td>
<td>28</td>
</tr>
<tr>
<td>18</td>
<td>OAuth 2.0 RCF</td>
<td>30</td>
</tr>
<tr>
<td>19</td>
<td>Risk of vulnerabilities</td>
<td>31</td>
</tr>
<tr>
<td>Chapter</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>20.</td>
<td>Login (as-is)</td>
<td>33</td>
</tr>
<tr>
<td>21.</td>
<td>Authorization (sequence)</td>
<td>34</td>
</tr>
<tr>
<td>22.</td>
<td>Authorizing with refresh-token (as-is)</td>
<td>35</td>
</tr>
<tr>
<td>23.</td>
<td>Sequence (with refresh token)</td>
<td>36</td>
</tr>
<tr>
<td>24.</td>
<td>Access vs Refresh token</td>
<td>38</td>
</tr>
<tr>
<td>25.</td>
<td>Target rank</td>
<td>39</td>
</tr>
<tr>
<td>26.</td>
<td>Broken access (sequence)</td>
<td>41</td>
</tr>
<tr>
<td>27.</td>
<td>Authorizing with OTP</td>
<td>42</td>
</tr>
<tr>
<td>28.</td>
<td>Authorization workflow (as-is)</td>
<td>42</td>
</tr>
<tr>
<td>29.</td>
<td>Authorization workflow (to-be)</td>
<td>44</td>
</tr>
<tr>
<td>30.</td>
<td>Authentication-Authorization in depth</td>
<td>45</td>
</tr>
<tr>
<td>31.</td>
<td>Access-Token from token end point</td>
<td>46</td>
</tr>
<tr>
<td>32.</td>
<td>Access-Token using refresh-token</td>
<td>48</td>
</tr>
<tr>
<td>33.</td>
<td>Access-Token using refresh token (expanded)</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>Section 1</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>Section 2</td>
<td>51</td>
</tr>
<tr>
<td>34.</td>
<td>Authorization (no malicious software)</td>
<td>52</td>
</tr>
<tr>
<td>35.</td>
<td>Authorization (malicious software trying to intercept)</td>
<td>53</td>
</tr>
<tr>
<td>36.</td>
<td>Authorization (malicious software with refresh-token)</td>
<td>54</td>
</tr>
<tr>
<td>37.</td>
<td>Authorization (resource owner with OTP)</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>Section 1</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>Section 2</td>
<td>57</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

1.1 Problem Statement
In the past, our focus was primarily on client applications such as Windows Forms or WPF, along with server-side web applications. These applications were not service-based and often followed the N-Tier architecture, relying on Windows credentials for identity-related information. Users would log into their company domain on their machines, providing sufficient authentication.
For internet web applications, Windows Authentication was common, and for outward-facing web applications, forms authentication was employed with a login screen at the application level. Additionally, applications frequently resided on the same domain, allowing a cookie with an authentication ticket to be shared for basic single sign-on when necessary.

In the present interconnected digital landscape, RESTful APIs (Representational State Transfer Application Programming Interfaces) play a crucial role in facilitating data exchange between diverse systems and services. While these APIs offer advantages in simplicity and scalability, they also pose significant security challenges. Existing security measures may prove inadequate to protect RESTful APIs from evolving threats, potentially resulting in data breaches, unauthorized access, and service disruptions.

This thesis aims to investigate and propose comprehensive security solutions to address the following key challenges in the context of RESTful API security, like:

- Current authentication and authorization mechanisms often fall a little short (little hole is big enough in security) in providing robust protection. Effective methods of authenticating users and authorizing web clients to access API resources need to be explored and studied thoroughly.
- Token-based security should not be handled by home-grown services, as the implementation is time taking, very risky, and needs updates frequently as behaviors of threats change every second.
• Refresh-Tokens, as are bearer tokens, once hackers (malicious software) get their hands on them, it is inevitably easy to get access-token to use and access data and breach system.

1.2 History
REST did not emerge in isolation; instead, it evolved from a specific context with defined objectives and underlying assumptions. Roy Fielding, who coined the term in his doctoral dissertation, contributed to a broader narrative rooted in the development of the "world wide web" (Fielding, 2000). The origins of the web trace back to the early 1980s at CERN in Switzerland, where Sir Tim Berners-Lee initiated the process. His initial system, Enquire, aimed to organize diverse projects and computers without network connectivity but interconnected documents through hyperlinks. Berners-Lee's vision of a globally linked information space persisted as he explored the internet, leading to the creation of the first browser and editor, aptly named World Wide Web. Notably, the inaugural www program was written in objective C for UI OS developers.

The key thing to note is, there were three novel innovations that formed the web. In order of importance:

a. URI
b. HTTP
c. HTML

URI, or as it was back then called Universal Document Identifier (UDI) was considered the most important because it was the only thing Sir Tim Berners-Lee thought required conformance by all the web's participants (Masse, 2012).

Fielding developed REST not as a standalone prescription for building APIs, but rather as a means to articulate the rationale behind the existing design of the web. His intention was to create a framework for assessing various proposals, ensuring compatibility with the web's architecture, and avoiding disruptions. REST evolved alongside the web, and over time, these ideas found their way into his dissertation. Subsequently, his work sparked numerous debates and discussions in
the form of blog posts. Crucially, REST is not merely a guideline for API development; it serves as a description of a comprehensive system architecture, notably that of the World Wide Web.

1.3 Definition

1.3.1 REST

- REST, operates as architectural style.
- The server follows a standard approach to respond to requests involving the creation, reading, updating, and deletion of recordings.
- REST is stateless, meaning every interaction is independent.
- It is an architectural style, which means it is a named set of constraints.

When a client initiates resource request using REST API, the resource server sends response of the current state of the resource at the server in a standardized representation (like, JSON or XML) to the client. This approach encourages developers to contemplate the proper behavior of well-designed web applications. Ideally, such applications should function as integral parts of a network of web resources, often referred to as a virtual "state-machine." Within this network, users navigate through the app by selecting resource identifiers or specific URLs and applying HTTP operations such as POST and GET, which are termed as applicable state transitions. The representation of the resource is subsequently transmitted to the end user, signaling the next application state.

REST is an architectural style made up of several other styles with some additional constraints added on top. It is also a named set of constraints on how components interact in a system with the intent of explaining the web architecture (Masse, 2012).

1.3.2 Representational State Transfer

While REST operates in a stateless manner, its name involves state transfer. Representational State Transfer, or REST, focuses on transferring "representations". Essentially, when we access a web page via a URL to view a server-based image file (which is referred to as RESOURCE), the server displays (presents) the image (REPRESENTATION) to the client. This representation
conveys the server's constant resource state, like the image file stored in the database, in a format understandable to the client, altering application state of the client. Despite the resource state remaining constant for clients, the representation changes, impacting the application state. In essence, REST refers to the implicit alteration of the server and client states through the presentation of a resource to the client.

![Diagram](image)

**Figure 1. Requesting Ethiopian Food (a state).**

### 1.3.3 Stateless

Every request must contain all the necessary information for its processing in REST APIs, eliminating the need for server-side sessions. Server applications are prohibited from retaining any data linked to the client's request. When a web page is opened in a browser, the requester acts as a service consumer, and the www server serves as the service provider. In a typical connection, both the client and server shake hands, establishing a TCP connection or session. Subsequently, the state transitions to ESTABLISHED, IDLE, TIMEOUT, etc., based on their behaviors. However, in REST, utilizing the stateless HTTP protocol means the server refrains from storing any session data about the client. The client bears the responsibility of providing all necessary details to the server for effective servicing, ensuring that when the URI is invoked, it contains sufficient client information to fulfill the requested resource (Masse, 2012).
Figure 2. Stateless.

Figure 3. Requesting Food (stateless).
1.3.4 API
An API, short for Application Programming Interface, serves as a standardized method for exchanging data between computers. To illustrate, when incorporating Google Maps into a website and accessing maps stored on Google's servers, a request is made to Google via an API. This API guides us on the specific web addresses to use when requesting the desired map data. To put it formally, a request is dispatched to the remote server to obtain a resource. In the analogy provided, the waiter functions akin to an API, facilitating the exchange of requests and responses between the kitchen and the customer (James & Daniel & Mathew, 2022).

![Figure 4. REST API Architecture.](image)

1.3.5 Client
A client, in the context of an API, refers to a program that issues requests on behalf of a resource owner (end user) to the API, seeking information or initiating changes in the application. For instance, a web browser acts as a client by interacting with various websites' APIs to fetch page content. Subsequently, the retrieved information is sent back to the browser and presented on the screen.

1.3.6 Request
This is a message dispatched to a server, soliciting an API to furnish a specific information service. In addition to the command, a request incorporates data (referred to as the body), typically associated with HTTP methods like POST, PUT, and PATCH. This data encompasses the details and representation of the resource to be created or updated.
1.3.7 Response

Response is data or information sent back to the requester after a request is made. Each response is associated with HTTP response codes like, 200, 201, 400, 401, and 500. Basically, there a 1xx, 2xx, 3xx, 4xx, and 5xx status codes.

- Informational: 1xx
- Success: 2xx
- Redirection: 3xx
- Client Errors: 4xx
- Server Error: 5xx

1.3.8 Resource

Resource refers to any data that the API can furnish to the client. For example, within Google's API, a resource could represent a user, a map, or a post. Each resource possesses a distinctive name known as the resource identifier. A set of these individually addressable resources, referred to as Resource Indicators, specifies the target services or resources to be requested, typically presented as a URI format variable representing the resource's identity.

1.3.9 REST Server

The REST API server, employed by the application handling client requests, houses the resources sought by the client. Through its API, the server engages with clients, maintaining a layer of interaction without granting direct access to the content stored in the database.
1.4 Applications Architecture
Applications no longer live in the same domain, and sometimes they are not even under our control. We started creating applications that integrated with different APIs. A client application could talk to its own API, but at the same time could talk to Google’s API, for maps integration, to name one. And those client applications changed as well. We are no longer building only server-side applications that communicate with APIs, we are also building single page applications like Angular, we are building mobile applications that communicate with APIs, and we are building APIs that communicate with another APIs (James & Daniel & Mathew, 2022).
Figure 6. Modern Architecture of Interacting Systems.

All these mean we can no longer protect our APIs by keeping them inside of the company walls. These APIs must be public; thus, it cannot be secured with forms authentication anymore, a mobile application doesn’t even have a notion of that. And a server-side web application doesn’t necessarily talk with in-company APIs, we might decide to host the APIs outside of the company walls and monetize it or the application itself might need to access to other APIs, so we are a bit stuck here. Old architectural approaches don’t work anymore because they were not designed for these above-mentioned scenarios. So, we started sending username and password over the wire on each request from client to API, and that quickly proved to be a bad idea. Once a packet sniffer gets access to the username and password, the packet sniffer will have even more access and can cause severe damage.
1.5 Storyline

1.5.1 Connectify’s Case

In the heart of Silicon City, a bustling metropolis filled with tech startups and digital innovation, a young and ambitious software engineer named Alex was tasked with a challenging project. Their mission: to secure the RESTful API of a popular social media platform, Connectify, by implementing OAuth 2.0 and OpenID Connect. Little did they know that this journey would take them down a perilous path of unforeseen security issues.

Alex started with high hopes and an eagerness to learn. They studied OAuth 2.0 and OpenID Connect diligently, understanding the protocols and their role in securing API access. Armed with knowledge, they began implementing these frameworks into Connectify's API.

As the implementation progressed, Alex discovered a series of vulnerabilities lurking beneath the surface. One of the most significant issues was the misuse of tokens. Connectify's API was issuing long-lived access-tokens without proper mechanisms for token refresh or revocation. This oversight could allow malicious actors to maintain access indefinitely. Additionally, Alex uncovered flaws in the authorization code flow of OAuth 2.0, which could lead to unauthorized access to user accounts. They realized that insufficient validation of redirect URIs could potentially redirect users to phishing sites, compromising their credentials.

1.5.2 Big Firms

API security incidents pose significant threats, with a recent survey revealing that 41% of organizations encountered such issues in the past year (2022). Another survey indicates a staggering 68% increase in API security incidents within a single year. The consequences of API breaches can be catastrophic, potentially exposing millions of sensitive user records and leading to severe trust issues between developers and users, along with financial repercussions.

Amongst API breaches, some list of the most ones of the last two years happen to the following major organizations and firms.

- Twitter (now known as X)
- Optus
1.6 Authentication and Authorization

Authentication Verifies that a user is who they claim to be, verifying a user who they say they are. For instance, when we check in to a hotel, we provide a driver’s license or a passport to show proof of reservation, to authenticate our identity, to show who made the reservation. Authorization determines if a user is allowed to perform certain actions or access certain resources or data. For example, when one arrives at the airport, they show a boarding pass to the flight crew members, so they can authorize the customer to board the flight and allow to hop on to the plane, and then based on the level of pass we sit either in business or economy class (Jaimandeep & Naveen, 2022).

Figure 7. Authentication and Authorization.

1.7 Token

A token serves as a digitally encoded signature employed for authentication and as well as authorization of a user to access and make use of specific resources within a network. Generated
by an authentication service, tokens encapsulate information that enables users to verify their identity without the need to enter login credentials. Upon the conclusion of a user's browsing session and their subsequent log-out from the service, the granted token undergoes destruction, ensuring the security of access.

In essence, OAuth 2.0 focuses primarily on authorization and access control. This framework empowers applications to make request and acquire access to resources on behalf of clients. OpenID Connect extends the functionalities of OAuth 2.0 by incorporating user authentication and identity information. This extension renders it apt for scenarios requiring both authorization and authentication, like single sign-on (SSO) solutions and resource owner authentication within contemporary web and mobile applications (Yvonne & Abhishek, 2023) (Gomi) (Jaimandeep & Naveen, 2022).

1.7.1 ID Token
The OAuth 2.0 ID Token is one of the many JWTs (JSON Web Tokens) that contains user authentication information, issued by STS (Secure Token Service) or authorization server during the OAuth 2.0 flow. It typically includes details like user ID, issuer, expiration time, and may be used by the client application to verify the resource owner's identity (Yvonne & Abhishek, 2023) (Jagdish & Kusum & Atulya, 2021).
1.7.2 Access-Token

Access-Tokens serve as the means by which applications initiate API requests on behalf of users, signifying the authorization grant to a specific application for accessing particular resource or user data. For client application, access-token is treated as an opaque string, utilized in an HTTP request without requiring an understanding of its content. It is crucial to maintain the confidentiality of access-tokens during transmission and storage, restricting visibility to the application, STS (authorization server), and resource server. The storage of access-tokens must be secured from access by other applications on the same device. Additionally, access-tokens can only be employed over HTTPS connections to prevent interception by third parties on non-encrypted channels. The token endpoint is where applications request access-tokens for users, involving verification, response handling, and error management (Yvonne & Abhishek, 2023) (Jaimandeep & Naveen, 2022).
1.7.3 Refresh-Token

Refresh-Token is a credential used to be granted a new access-token when the one in use expires. It allows a client application to maintain ongoing access to a user's resources without requiring them to re-enter their credentials. The refresh-token is securely stored and exchanged for a new access-token as needed, extending the validity of the authentication (Yvonne & Abhishek, 2023) (Jaimandeep & Naveen, 2022).

1.8 Authentication and Authorization Protocols

OpenID Connect (OIDC) and OAuth 2.0 serve as widely adopted protocols in the industry for user authentication and authorization. These protocols work hand in hand, outlining the procedures for authenticating users by a server and subsequently authorizing their access to resources:
• OpenID Connect (OIDC) is constructed on top of OAuth 2.0 and stands out as a prominent authentication standard. It introduces an ID token type, complementing OAuth 2.0 access-tokens and refresh-tokens. OIDC also regulates aspects that OAuth 2.0 leaves to will and discretion, including endpoint discovery, scopes, and the dynamic registration of clients. OIDC builds upon OAuth 2.0 by incorporating user authentication and Single Sign-On (SSO) capabilities. It facilitates the retrieval and storage of authentication details for end users while also introducing various OAuth 2.0 scopes for applications to access user profile information (OIDC, 2014) (Yvonne & Abhishek, 2023) (Jaimandeep & Naveen, 2022).

• OAuth 2.0 manages and delegates authorization for accessing protected resources, such as web apps, native apps, or API services. It enhances API security through the use of scoped access-tokens (OAuth 2.0, 2012) (Yvonne & Abhishek, 2023) (Jaimandeep & Naveen, 2022) (Meng-Yu & Tsem-Huei, 2013).

1.9 Authentication Service
The Authentication Service operates by soliciting information from an authenticating party and verifying it against the configured identity repository through the designated authentication module. Upon successful authentication, the user session is initiated.

1.10 Authentication-Authorization Flow
Authentication and Authorization are most important types of validation, not something we typically want to do ourselves in our application, rather the different types of middleware take care of that. Important to know is that validation procedures are different when using different flows, and they can include additional steps if, for example, additional encryption is used outside of TLS. Next to that, some parts of it are optional and are not implemented by all identity providers or clients, and the other way around is possible as well. Identity providers and clients
can decide on additional validation steps, so not every IDP or client will use the same procedures. We will focus on what is required for the authorization code flow in the way that we set it up. We’ll start with validation of the identity token at the level of the client, and then we'll cover how the access-token is validated when it reaches the API.

As mistakes are inevitable, and keeping home-grown systems is not easy either as new vulnerabilities are regularly discovered. We are trying to solve two issues here, no longer handling user credentials at the application level and finding a way to ensure that the tokens are safe enough to be used for authentication and authorization for all different types of applications. Thus, we need a proven protocol that handles common concerns (Yvonne & Abhishek, 2023) (Jaimandeep & Naveen, 2022) (Meng-Yu & Tsem-Huei, 2013).

Figure 10. Sequence diagram, authentication-authorization on first time login (as-is).
1.11 OpenID Connect vs. OAuth 2.0

OpenID Connect (in short OIDC) and OAuth 2.0 are two distinct, yet interrelated protocols employed in contemporary web and mobile applications for authentication and authorization purposes. Let’s delve into each protocol (OIDC, 2014) (Yvonne & Abhishek, 2023).

OpenID Connect (OIDC), it functions as an identity layer built upon OAuth 2.0. OIDC enhances the OAuth framework by introducing authentication capabilities, allowing the client to verify the identity of the end-user. OIDC provides identity information about the authenticated user in the form of ID tokens. The core components of OpenID Connect encompass:

- **ID Token**: it’s a JWT (JSON Web Token) containing claims about the authenticated user, such as their unique identifier and user information.
• **UserInfo Endpoint**: An API endpoint that the client can use to obtain additional user information.

• **ID Token and UserInfo Claims**: These claims can include the user's name, email address, and other profile information.

• **Discovery Endpoint**: A URL where the client can discover metadata about the OpenID Connect configuration, such as supported endpoints and public keys.

• **Relying Party**: This refers to the client application requesting an ID token.

• **OpenID Provider (Identity Provider)**: The authorization server responsible for issuing the ID token.

• **ID Token**: This token, issued by the OpenID provider, encapsulates information about the end user in the form of claims.

• **End User**: It is the resource owner and the user's information are contained within the ID token.

• **Claim**: A claim denotes a piece of information about the end user the client tries to access.

OAuth 2.0 is a framework for Authorization enabling a client application to acquire limited access to a user's resources on a separate resource server without exposing the user's credentials. While not an authentication protocol in itself, it is commonly used alongside authentication mechanisms (OAuth 2.0, 2012) (Sakimura, n.d.) (Yvonne & Abhishek, 2023).

The pivotal components of OAuth 2.0 include:

• **Client**: The application seeking access to a user's resources.

• **Authorization Server**: authenticates the end user and provides and grants access-tokens to the client on behalf of the resource owner.

• **Resource Server**: The server housing the user's resources that the client intends to access.
• **Authorization Code**: A temporary code obtained by the client post-user authentication, utilized for exchanging tokens.

• **Access-Token**: this is a token granted by the authorization server’s token end point to the client, enabling access to the user's resources.

In summary, OAuth 2.0 is primarily focused on authorization and access control, allowing APIs and web applications (and/or mobile apps) to request and get permission access to resources on behalf of clients. OpenID Connect built on top of (extends) OAuth 2.0 to provide functionality of user authentication and identity information, making it suitable for scenarios where both authorization and authentication are required, such as single-sign-on (SSO) solutions and user authentication in modern web and mobile applications (Yvonne & Abhishek, 2023).

![Figure 12. Authentication-Authorization.](image-url)
1.12 Authorization-Code-Flow with PKCE

The flow Authorization Code stands as one of the four standard OAuth 2.0 grant types, employed to securely acquire access-tokens. Predominantly utilized in web applications, it offers a more robust security mechanism for obtaining access-tokens compared to the Implicit Flow (Yvonne & Abhishek, 2023).

Let's explore how the Authorization Code Flow operates:

1. **User Initiation of Authorization:** The user engages with a web application (the client) and triggers the "Login" button. The client redirects to the STS’s (authorization server's) authorization endpoint, conveying essential client configuration values, like client ID, requested scope, and a redirect URI where the resource owner will show (the client sign-in redirect page) after successful authentication. This process incorporates PKCE (Proof Key Code Exchange) which the client generates upon initial request.

2. **User Authorization Grant:** On the authorization server's account login interface, the user undergoes a login form or alternative authentication method, like Microsoft login page. Following successful authentication, once PKCE (code_challenge) is stored, the user is prompted (optionally) to approve the web client's access request. If consent is given (optionally), the authorization server redirects the user back to the client's designated sign-in redirect URI, incorporating an authorization code as a query parameter.

3. **Web Client requests for Tokens using Authorization Code:** Upon receipt of the code (authorization code, granted by the STS), the client application initiates a secure, server-to-server request to the authorization server's token endpoint. Request payloads include the authorization code, client credentials (client ID and client secret), and the redirect URI, validating the code against the one used in the initial request. Upon successful verification, the authorization server responds with an access-token and a refresh-token.
4. Client Utilization of Token: Armed with one of the tokens granted, the access-token, the client application gains the ability to make requests to protected resources (APIs) on behalf of the user. The access-token is included in the Authorization header of API requests, enabling seamless interaction with protected resources.

**Key Points:** The Authorization Code Flow is deemed more secure as the access-token remains shielded from the resource owner's browser or front-end JavaScript, being securely exchanged on the server-side. Safeguarding the client secret, employed for authenticating the client with the authorization server, is imperative. The acquired access-token typically has a short lifespan, and for prolonged access, the client can use the refresh-token without user interaction. Implementing HTTPS at all stages of this flow is vital to ensure data confidentiality, making it well-suited for web applications requiring high security (Lodderstedt & Bradley & Labunets & Fett, n.d.).

![Diagram](image_url)

**Figure 13. Authorization code flow.**

### 1.13 Code Verifier

In OAuth 2.0 Authorization Code Flow, a code verifier is a string that the client application creates before initiating the authorization request. It is used as a part of a security mechanism to ensure the integrity and confidentiality of the authorization process (Yvonne & Abhishek, 2023).

Here's how it works:
1. Client Generation: The client application generates a code verifier, typically a random string of a certain length. This code verifier is unique for each authorization request.

2. Code Challenge: The client application then hashes the code verifier using a specific method, often SHA-256, to create a code challenge.

3. Authorization Request: Upon commencing the authorization process, the client incorporates the code challenge method (typically "S256" for SHA-256) and the code challenge itself into the process of authorization request dispatched to the STS (Secure Token Service).

4. Authorization Server Validation: Upon receipt of the request, the authorization server employs the code challenge method and the code challenge to authenticate the integrity of the request. It validates that the code challenge corresponding to the original code verifier. Through the utilization of this code verifier and challenge mechanism, the Authorization Code Flow augments security on multiple fronts:

   **Confidentiality:** It ensures that the code verifier remains confidential because only the client knows it. This helps protect against interception of the verifier during the authorization process.

   **Integrity:** The code challenge helps verify that the authorization request hasn’t been tampered with during transmission. If the code challenge doesn’t match the original verifier, the request is considered invalid. This added layer of security helps protect against certain attacks, such as code injection attacks or eavesdropping, and ensures that the authorization process is more robust and trustworthy.
Figure 14. Handing authorization token.
Figure 15. Authorization code flow with PKCE.
CHAPTER 2
RELATED WORKS

When we talk about front-end, we talk either directly or indirectly about JavaScript, that's essentially how things are built in browser. We may be using some other language that eventually transpile into JavaScript anyway. We have to account for evil JavaScript ending up in an application, this can happen in various ways, we can install a malicious dependency from npm, for instance. It's basically like bingo, at this time 50% chance of getting the right one or the wrong one. Like, if we include remote code files or the likes, if we load some file from a third-party service provider, we might get attacked and we end up getting malicious code in our application, or we have cross-site scripting, where the attacker provides data to the application and there's a mistake and that data becomes executable code.
Malicious JavaScript remains a big threat with all the measures we have in place, it's still something we have to account for. Cross site scripting, token exfiltration, sounds super fancy in essence it steals tokens, access-tokens, and refresh-tokens (Phillippe, 2023).

2.1 Cross Site Scripting Scenario
Restograde, it's a restaurant review app very simple very straightforward it handles reviews from people all over the world. And the app is just running some OAuth, use the authorization code
flow with PKCE, the app typically stores tokens somewhere, say at local storage session, the app needs them to attach on outgoing requests to the API, which is a common scenario, the question is what happens when the attacker shows up, i.e. a piece of malicious code ends up in there and the attacker now executes code inside the application, that's how cross-site scripting works, the attacker reads that storage and get all the data out of storage and send it off to a server controlled by the attacker, just a simple get request or a post request. The malicious server then uses the access-token to contact an API in the name of the user, that is token exfiltration (Phillippe, 2023).

### 2.2 Refresh-Token Rotation

There's a whole bunch of countermeasures to try and prevent token exfiltration from causing trouble. One of the guidelines is we should keep our access-tokens short-lived because if the access-token that's used is only valid for 5 or 10 minutes then the damage the window of abuse is limited to 5 to 10 minutes, that's good especially because those tokens are typically jwts, Json web tokens, that are not revocable or there's no infrastructure to revoke them during their lifetime. The second guideline we find in the spec is like we have to use refresh-token rotation, because bearer tokens are not very secure meaning, that if the attacker gets hold of a refresh-token, they can now use it with the authorization server to get new fresh tokens.

The malicious software gets access-token-1 and refresh-token-1, and access-token-1 is valid for 10 minutes and then access-token 1 expires after 10 minutes, the app needs to get a new one and it's going to use refresh-token-1 to get access-token-2, and it also gets refresh-token-2, after the next nine minutes access-token-2 expires so the app uses refresh-token-2 to get access-token-3 and refresh-token-3. And then if access-token-3 is about to expire, refresh-token-3 is used to get access-token-4 and refresh-token-4, this goes on and on until some kind of absolute lifetime has reached. If the authorization server sees a refresh-token twice it assumes that something fishy is going on because the legitimate app is not going to decide halfway through, like use refresh-token-2 from 10 minutes ago, to use that one instead of the new one.
So, what is the idea here? If the attacker comes in and steals refresh-token-2 at some time and they use it, they will get access-token-3 and refresh-token-3. That’s a red flag, they now have access to a REST API data access in the name of the legitimate client along with the refresh-token, but the client app is unaware that this happened. So, after a while the app is about to use refresh-token-2 as well, and the authorization service will say, “no I’ve already seen this refresh-token, I’m going to assume somebody else has used that in your place”, so it kills the session. Hence, refresh-token-3 becomes invalid. In the future the attacker no longer has access to that token chain as all the tokens are invalidated, basically one way of securing refresh-tokens and so REST APIs access control. That’s a good security measure because now even if a token is stolen, we can prevent it from being abused in the long term.

Access control is a fundamental concern in modern application development, ensuring that only authorized entities access sensitive resources. In many systems, Identity Providers (IdPs) play a pivotal role in authentication and authorization processes. Understanding the lifecycle of refresh-tokens, including issuance, expiration, and revocation, is crucial. Proper management ensures that access remains secure and up to date. refresh-tokens can be exchanged for access-tokens, which grant clients access to protected resources. API servers should validate access-tokens.
before allowing resource access. Access control is often granular, involving different scopes and permissions. APIs should honor these scopes to ensure fine-grained authorization. Implementing token rotation mechanisms help mitigate security risks associated with long-lived refresh-tokens, reducing the window of vulnerability. To prevent abuse and protect APIs, rate limiting and throttling mechanisms should be in place, restricting the number of requests a client can make. refresh-tokens must be securely stored and transmitted. Techniques like encryption and proper storage mechanisms are vital to prevent token leakage. Comprehensive logging and audit trails are essential for monitoring and investigating access patterns and potential security breaches. Implementing mechanisms for token revocation and cleanup ensures that lost or compromised tokens do not pose a persistent security threat to some extent (Phillippe, 2023) (RFC 6749, 2012).

2.3 RFC 6749
Authorization Code Flow defined in RFC 6749 (Request for Standard):-
RFCs define OAuth 2.0 and the use of bearer tokens including access and refresh-tokens. They provide detailed specifications for token-based authentication and authorization. OIDC (OpenID Connect) is an identity layer built on top of OAuth 2.0. It provides additional features, including ID tokens, which can be used to enhance security and user authentication. Refresh-Token Rotation is a technique used to enhance security by regularly rotating refresh-tokens. It helps mitigate the risk of long-lived refresh-tokens being compromised. Token Storage and Encryption, securing storage and encryption of tokens, including refresh-tokens, are crucial for protecting user data (RFC 6749, 2012).
Within the domain of web application security risks, **A01:2021-Broken Access Control** has undergone a significant shift, ascending from its fifth-place position to now claiming the top spot as the category with the highest severity. Data analysis reveals that, on average, 3.81% of tested applications manifested one or more instances of Common Weakness Enumerations (CWEs), totaling over 318,000 occurrences within this specific risk category. Among the diverse risk categories, the 34 CWEs associated with Broken Access Control have surpassed others in terms of frequency within applications.

Refresh-Tokens are fundamental part of authentication and authorization systems, but they come with certain challenges and potential problems, like Long-Term Security Risks. Refresh-Tokens typically have a longer lifespan than access-tokens. This can pose a security risk if they are compromised or stolen since an attacker could use them to obtain new access-tokens repeatedly. As said above briefly, best practices include token rotation, and refresh-tokens need
to be securely stored and managed. If they are not properly protected, they can be vulnerable to theft, potentially leading to unauthorized access (OWASP, 2021).

2.4 Challenges

2.4.1 Revocation

Revoking refresh-tokens can be challenging. Unlike access-tokens, which expire relatively quickly, refresh-tokens often have longer lifetimes. Revoking a compromised refresh-token can be more complex, and there may be a delay before it takes effect. Understanding how to revoke access and refresh-tokens when they are no longer needed or when a user logs out is an important aspect of access management. Token-Based Authentication Libraries, various programming languages and frameworks offer libraries and tools for implementing secure token-based authentication. Following Industry Best Practices, keep an eye on industry best practices and guidelines published by organizations like OWASP (Open Web Application Security Project). They often provide valuable insights into access management and security.

2.4.2 Token Leakage

If a refresh-token is leaked or exposed in logs or other insecure places, it can be used by an attacker to obtain new access-tokens, even if the original access-token has expired. Token Rotation Complexity, implementing secure token rotation strategies to mitigate security risks can be complex and require careful design and development.

Figure 19. Rank of vulnerabilities (the lower the rank the less secure it is).
2.4.3 Others

Scalability, as the number of users and clients increases, the management of refresh-tokens can become challenging. Storing and validating a large number of refresh-tokens efficiently can be resource intensive.

Privacy Concerns, refresh-tokens often contain user-specific information. Storing these tokens may raise privacy concerns, especially in jurisdictions with strict data protection regulations.

Session Management, maintaining consistent user sessions across various devices and platforms can be tricky when relying on refresh-tokens. Ensuring a smooth user experience requires careful session management.

Token Expiration and Renewal, deciding when to expire refresh-tokens and how to renew them can be challenging. Too short a lifetime can lead to poor user experience, while too long a lifetime increases security risks.

To address these challenges, it's crucial to follow best practices in token management, including secure storage, proper revocation mechanisms, thorough auditing, and regular security assessments. Additionally, considering alternative authentication methods, such as short-lived access-tokens with dynamic authorization, can help mitigate some of the risks associated with refresh-tokens.

2.5 When to use Refresh-Token

The primary aim of incorporating a refresh-token is to significantly diminish the lifespan of an access-token. This refresh-token, subsequently, serves to authenticate the user as necessary for the application, mitigating issues such as cookie blockages. In practical terms, when a browser initiates a request to a REST API endpoint for a resource, exclusive to authenticated users, the web application necessitates the user's credentials for each request. Once the user authenticates (logs in), the application on the user's browser gains access to the resource by sharing an access-token. This token streamlines subsequent API calls from the browser, mandating credentials, without requiring the user to repeatedly log in. During the process of sharing the access-token, the system may also furnish a refresh-token. So, refresh-token becomes instrumental in
authenticating the user for subsequent API calls, even in the event of the access-token expiring, by soliciting a new access-token. Consequently, refresh-tokens empower applications to acquire fresh access-tokens through uncomplicated API calls, obviating the need for users to authorize cookies or undergo multiple login procedures.

Figure 20. Existing system authentication-authorization (login for the first time: as-is).
2.6 Drawbacks

It's also accurate that we might not necessarily require the "additional capabilities" provided by refresh-tokens to maintain smooth user sessions. Cookies and silent authentication methods have their own advantages. Consider a situation where refresh-tokens could pose a serious threat to an application. If a refresh-token is compromised or either stolen or accessed by someone else, the intruder not only gains API resource access but also extends the granted access time. This is a concerning scenario for both developers and users. However, precautions like refresh-token Rotation and Automatic Reuse Detection can mitigate these risks, emphasizing the advantages of refresh-tokens. In these approaches, when a refresh-token is used to access resources, the system not only provides an access-token but also issues a new refresh-token.
Subsequent API requests can then use these updated refresh-tokens. If an older refresh-token is used in a request, it is efficiently rejected, assuming the requester is unauthenticated.

What if the legit client goes offline (resource owner hopes on to an airplane, internet goes down), or stops requesting for an access token with the stolen refresh token, the malicious software will have indefinite access to the resource until the client comes online, and what if the client access is compromised to the level it can’t access its own resource (OWASP, 2021) (Basel & Nishu).

Figure 22. Authorization using refresh-token (client already authenticated: as-is).
Figure 23. Sequence diagram using refresh-token (client already authenticated: as-is).
When security is implemented, there are a number of security considerations, like Transport Protection, Cross Origin Resource Sharing (CORS), Cross Site Request Forgery (CSRF), and Cross Site Scripting (XSS). We should make sure all are covered in our applications, mainly authentications and authorizations. Authentication, especially in public clients, being superficial and/or as they do not process data directly, to some extent, Authorization is by far the area where we have to be careful and particular about, as it is where we place on top of an access point where the data is pulled from.

As other problem considerations, which are mentioned above, are handled by the OAuth 2.0 protocol, it’s not something we want to take care of ourselves or spend our times on, we just have to apply and go with a well-defined protocol. And transport protection refers to how we protect requests and responses that are made between browser and backend resources (REST API) that our application talks to. This also applies to any backend to backend communication between services, we just have to use https everywhere. The underline https protocol is often referred to as SSL (Secure Socket Layer), which is now referred to as TSL (Transport Security Layer).

Refresh-Tokens boast a prolonged validity compared to access-tokens. As an illustration, an authorization server issues a short-lived token (JWT) with a 15-minute expiration, alongside a refresh-token embedded in HTTP cookies, extending its lifespan to 7 days and even sometimes to a year or so. This refresh-token serves the purpose of obtaining a renewed access-token through the refresh-token grant flow. To initiate the refresh of both access-token and an ID token, request of a refresh-token is dispatched with a grant_type of refresh_token. Additionally, when aiming to refresh the ID token, the openid scope is included. Upon validation of the refresh-token, the system responds by furnishing a fresh access-token along with a refreshed (new) refresh-token.
Life span of refresh and access-token are significantly different, refresh-token may live up to years and in some cases never even expire. Whereas access-token in most scenarios can live up to one hour and have to be replaced with a new one if client has to stay authenticated. One thing to note is that the use of https doesn’t stop refresh token exfiltration, as https secures the communication over a computer network, whereas exfiltration (XSS) happens within the domain of the client application (browser), and user’s access to their own account can be compromised and any kind of requests on behalf of the resource owner can be done, like impersonating (potentially logging out the user and even changing credentials).

To the current implementation of OAuth 2.0, we are adding another level of security, to the vulnerable area where the refresh-token is a long lived and can be used to get an access-token and a new refresh-token for future use, which seems to be an endless cycle until the refresh-token is left without use and expire. According to OWASP, an awareness document which organizations use as a de facto standard since its inception in 2023 as a coding or testing standard, Broken Access Control was the most serious web application security risk, and the
The proposed solution aims in mitigating this risk and make it one of the least security risks in the coming future.

In this proposed solution framework securing an application's authentication system relies heavily on safeguarding refresh-tokens. To explore this further, we may want to consider the following strategies and recommended practices:

- **Token Storage** - Explore diverse storage methods for refresh-tokens, including databases, secure key-value stores, or token vaults. Assess the security and performance implications associated with each storage approach.
- **Token Encryption** - Experiment with encryption algorithms and key management to enhance the protection of stored refresh-tokens. Consider incorporating hardware security modules (HSMs) for an extra layer of security.
- **Token Rotation** - Introduce token rotation policies, periodically invalidating refresh-tokens and replacing them with new ones. Experiment with different rotation intervals to strike a balance between security and usability.

![Figure 25. Target rank of our enhancement by 2024.](image-url)
- **Token Access Control** - Experiment with access control mechanisms to limit who can request and use refresh-tokens. Implement user consent features for users to control token issuance.

For clarification purpose, let’s see the as-is and to-be scenarios where the solution framework pinpoints the problem and proposes a solution by illustrating a sequence diagram and workflow of each scenario.

3.1 Authorization

3.1.1 AS-IS

The current workflow of granting access-token using refresh-token is pretty straightforward and it’s like a handshake: provide a refresh-token and, access-token and a new refresh-token is granted and exchanged. The security hole here is what if it’s another client or application which did not initiate the process is the one asking for an access-token using refresh-token on behalf of the legit client. Currently we are not doing anything about this vulnerability. Anyone with a refresh-token will be granted an access-token and in the worst case a new refresh-token for later use.

The red rectangle implies where the danger zone exists, a malicious software can, for instance do Cross Site Scripting (XSS) and steal the refresh-token until the STS is aware when the same refresh-token is used by the legit client.
3.1.2 TO-BE

What we want to do is involve the end user (resource owner) using multi factor authenticator. As refresh-token is a long lived token, the chance of a malicious software gets its hands on the token is also very likely. Even if the client stores tokens at a very secure location of session storage, it is highly likely a malicious software intercept and abuse tokens.

Thus, we are prompting the resource owner to provide a One Time Password (OTP) upon requesting an access-token, and if that checks out anew reset token and a access-token will be granted.

Figure 26. Broken access, authorization using refresh-token (as-is).
3.2 Workflow

3.2.1 AS-IS

The workflow shows the same scenario as in the sequence diagram, for an already authenticated client, it is not required for the user to interact with the system and provide any form of credential whatsoever. As long as the client has an active session at the STS level, the STS will provide the client with access-token and refresh-token. The REST API sends validation of access-token to the STS upon resource request. Once the REST API gets a go from the STS, it will process the request and responds.

As long as there is an active session, no matter what token the client sends along with the request token won’t be revoked, rather the client gets an un-authorized (401 code) response. In addition to the insecurity, this will create an overhead on the REST API to process a request with an invalid token, even though the response is always un-authorized.
3.2.2 TO-BE

The proposed solution enforces the involvement of resource owner (end user) upon requesting an access-token on refresh-token. As long as there exists an active session of the client, which is requesting the access-token, refresh-token will be validated and if that checks out the resource owner is required to provide a One Time Password (OTP).

A user is required to register their device (in this case a mobile device) at the Multi Factor Authenticator by installing one of the well-known Multi Factor Authenticator Apps, like Google or Microsoft Authenticator, same authenticator is configured and added at the STS level.

Upon requesting an access-token, the STS requires the resource owner to provide a One Time Password (OTP) and when provided will request an approval from the Authenticator that was configured and added at the STS. If that checks out, client app will be granted a new refresh-token and an access-token.
3.3 Authorization Code Flow: Client to be Authenticated

For an already authenticated client whenever an access-token expires, the client should request a new one by making a request to the STS (authorization server) along with refresh-token as a parameter. The authentication process is triggered by the client by redirecting the user to a login page at the authorization server (if no third party authenticator used). The user interacts with the authorization server, provides credentials and consent so that the client accesses REST API resources. Upon user consent (if consent is required), the authorization server issues an authorization code to the client. The client requests the authorization server, with payloads of the authorization code obtained in the previous step. The STS (authorization server) validates the
client and the authorization code. If that checks out, the client sends a token request to the token endpoint, including the authorization code. The authorization server verifies the code and issues an access-token and optionally a refresh-token to the client. The client uses the access-token to request access to the protected resources from the resource server.

### 3.4 PKCE

As tried to explain, when with PKCE, the client creates a code verifier and hash that to a code challenge. This is sent upon initial authentication request along with username and password. When a client asks for a token, it sends the code challenge to be compared against a hashed code verifier that was sent at the initial authentication request and stored at the STS.

![Sequence diagram of authorization code flow](image)

**Figure 30. Authorization code flow sequence diagram in depth, to be authenticated.**
3.5 Authorization Code Flow: Authenticated Client

3.5.1 AS-IS

If the client has an active session at the STS level and a non-expired access-token, there is no need for the user to provide credentials and/or the client doesn’t need to be redirected to the Authorization end point for to provide code challenge and get in return the authorization code. The STS just provides a new refresh-token to the token end point and an access-token through a back channel without requiring any redirect from the client to the STS and back to the client. Once the client grabs access-token it carries on sending a request to the REST API, then the REST API gets the access-token validated at the STS level before granting access.

![Diagram of Access Token Grant](image)

**Figure 31. Access token grant from token end point using refresh token (as-is).**

3.5.2 TO-BE

Here, even though the client has an active session at the STS level and that active session is something we check first before we proceed to the other processes in our approach, we are not
granting an access-token and a new refresh-token to the client before the resource owner (end user) provides a one time password (OTP). This way, we insure the legit client is active and it’s the one which is calling the rest call.

When a client requests the STS (Authorization Server) for an access-token, the token end point then checks the expiration of the access-token. If expired, the refresh-token is used to request an access token from the token endpoint.

In the to-be scenario (in the proposed solution), we have to make sure whether this request is coming from the original client or a malicious software abusing a token on behalf of the legit client. The scenario in a step by step fashion is:-

- The token endpoint redirect the call, instead of granting an access-token as in the as-is (current system), to the authorize endpoint.
- The authorize end point will utilize account/otp page and display that to the resource owner.
- The resource owner uses the authenticator which is registered at the STS level to generate a one time password (OTP).
- Once the OTP is generated, the resource owner enters that same OTP at the STS level
- STS will validate the OTP against the Authenticator.
- If the validation checks out, access-token will be granted to the client and, the STS account/otp page will redirect to the client page.
3.5.3 TO-BE (Expanded)

In the proposed solution (in more detail), after checking for an active session and for un-expired refresh-token, the end user will have to have an OTP to provide from the registered authenticator and provide that upon request.

If the OTP is not valid, all token under that client will be revoked and the resource owner is required to provide credentials, this way we can secure the REST API and block any kind of request calls with a previously existing session and/or tokens.
Figure 33. Access token grant from token end point using refresh token (to-be) expanded.
Figure 33. Section-1. Access token grant from token end point using refresh token (to-be) expanded.
Figure 33. Section-2. Access token grant from token end point using refresh token (to-be) expanded.
3.6 Solution

- **Case One:** A happy path, no malicious software.

![Diagram of authorization process](image)

Figure 34. Authorization (no malicious software).

- **Case Two:** A malicious software running behind the scene without getting hold of any kind of token at the moment, may be providing a feature (like a search functionality) at the client level for the resource owner to make use of.

The malicious software is about to perform Cross-Site Scripting (XSS) by injecting malicious scripts into our web client (web application) which is about to be executed or run by the end user and this user input could not be handled, validated, or sanitized by the web application feature properly or couldn’t be stopped. The injected script can then execute in the context of victim’s browser, potentially stealing sensitive information or performing unauthorized action on behalf of the user.
Case Three: For the malicious software to get the best out of refresh and access token, it has to wait, or keep on hearing the heartbeat of the client application about to be compromised, until it (the client application) goes offline. This, indeed, is the case that the malicious software to have hold of the refresh-token for longer period of time and cause a greater damage on the resource owner’s account who was using the web application.

Token rotation in OAuth 2.0 involves refreshing an access-token to maintain a user’s session without requiring re-authentication. This typically occurs through the utilization of a refresh-token. When an access-token expires, the client can send the refresh-token to the STS (authorization server) to grab a new access-token, allowing non-stop and/or continuous access without prompting the user for credentials again. It enhances security by limiting the exposure of long-lived access-token. Feature added is that a refresh-token can never be used twice, and if STS detects a reuse of refresh-token it revokes all tokens and will enforce re-authentication.

The question arises when the malicious software gets its hands on the refresh-token and keeps on waiting for the client app to goes offline:- malicious software will have unlimited access on the access-token, causing a greater damage.
Case Four: We have put the solution in place and the malicious software gets its hands on the refresh-token and waiting on the client to go offline. And we are showing on the next sequence diagram the scenario where the access-token is not expired and another scenario where access-token is outdated.

The legit client itself need to get a One Time Password (OTP) upon requesting an access-token using refresh-token and provide that by generating the OTP frm the registered authenticator. That will be verified at the STS level if the OTP is valid and un-expired.
If this all checks out the client app will be granted an access-token and a new refresh-token that can be utilized to get the next rotation value of refresh-token and a new access-token. Since the client is still online (active), even if the malicious software request an access-token using the refresh-token that it has possessed before the client makes use of that same refresh-token, after a while when the web client requests an access-token using the refresh-token that the malicious software has already used, the STS automatically reokes all the tokens, and halts the sesion it has with the legit client. Therefore the malicious software needs to wait until the web client goes offline, which the malicious software reads through the heart beat of the client.

Figure 37. Authorization (resource-owner provides OTP).
Figure 37. Section 1. Authorization (resource-owner provides OTP).
Figure 37. Section 2. Authorization (resource-owner provides OTP).
Case Five: The client is not active or it’s offline, say for instance the resource owner hops on to an airplane for a 4 hrs flight and now the malicious software is not getting any sort of heart beat from the client, right away the malicious software start abusing all forms of token, say for an Hr. and now should get a new access-token. It makes a request and that will be granted with no exception whatsoever.

Considering the solution of multi factor authentication on top of refresh-token rotation makes an ultimate enhancement in a way the malicious software can have the ..

And in terms of usability, we are not asking the resource owner to enter ..

Figure 38. Proposed solution validated (malicious software failed to get access token).
Figure 38. Section 1. Proposed solution validated (malicious software failed to get access token).
Figure 38. Section 2. Proposed solution validated (malicious software failed to get access token).
3.7 Flow Chart

![Flow Chart]

Figure 39. Proposed solution workflow.

3.8 Pseudocode

3.8.1 Check Session

The very first thing we do is check if the client has an active session or not, then either carry on with the access-token it has in hand or revoke the token and redirect to the client sign in page.
3.8.2 Check Access-token

With the current protocol of OpenID Connect token expiration is way faster than refresh-token, and if access-token is active no need of using refresh-token. So, before we check if we have an active refresh-token we should check if access-token is active, and if that checks out then it means we have an active refresh-token, as the STS always provides a new refresh-token along with an access-token.
3.8.3 Check Refresh-Token

Refresh-Token is used to get a new refresh-token and an access-token, as refresh-token are long lived by default, that’s where we want to add our security enhancement.

```java
endpoint checksession (cookie-session)
    if active-session
        check access token
        if active access token
            continue
        else
            check refresh token
            if active refresh token
                redirect to authorize end point
                redirect to account/otp page
                show an OTP page to the user
                if OTP checks out
                    redirect to client page
                else
                    redirect to token end point
                    revoke token
                    redirect to authorize end point
                    redirect to client login page
            else
                redirect to token end point
                revoke token
                redirect to login page
    else
        redirect to token end point
        revoke token
        redirect to authorize end point
        redirect to client login page
```

Figure 42. Check refresh toke.

3.9 Solution Implementation

Solution can be achieved with a request, starting at the authorize endpoint with a callback URL, that is the one time password (OTP) page. The parameter in the URL of prompt can be “otp”, which tells the STS to only proceed if it can complete the request if the OTP value provided checks out. Then it can immediately redirect back to the client’s otp-callback page, which
potentially can be the page the resource owner was at. So now, as long as the resource owner is alive (is active) and the client’s session with the STS is alive, access tokens will always be valid after OTP checks out. The key thing is to just ensure client side token expiration is less than the STS’s by some minimum margin.
4.1 Refresh-Token Rotation

With rotation of refresh-token in place, what if the STS received a request with that refresh-token only once, nothing happens, meaning the STS will allow provision of access-token and a new refresh-token. And if nothing happens and the attacker has that token, they can use it until the absolute lifetime is reached, let's say we log in now and lifetime is eight hours, we steal it in 30 minutes and that means I would have access of that system for 7 hours and 30 minutes. Well, it depends on what the application is, how bad the damage is going to be. But that's not supposed to happen so how would an attacker do that? Well very simple, here's the scenario the attacker does the same thing, steals the token, sends them off to their server and keeps track of them and then they just wait and wait and maybe they repeat step one and two if there's new tokens refresh-token two and three and four and they wait until the application goes offline. We wait until the application goes offline. When it's offline, this means we close the tab, like browser disconnected from the network and we are on a plane or on long vacation, and the attacker uses our latest refresh-token. The attacker resumes that refresh-token chain and has access for the remaining lifetime of that chain.

4.2 Exfiltration

That is a fairly trivial way to bypass refresh-token rotation and that's essentially a takeaway here. Things like this, they severely underrepresent the capabilities of malicious JavaScript because an attacker can do so much more. And that's essentially a very important message, the attacker can do anything a legitimate front-end application can do and they can write sophisticated JavaScript code.
We have a malicious server, the malicious server is basically running and it's going to accept tokens and abuse tokens when it wants to, basically it has a data endpoint to accept incoming data including the refresh-token. We also are going to launch an attack file from attack.js with the full payload to steal tokens from the application. Basically, we send a fetch request (a post request) and we include the data from session storage in the request and that's how we steal tokens. With the search feature, this is a cross site scripting vulnerability, so what is going to happen is, we load the app with the search term, send a link to the restaurant review app with search for like Ethiopian food in Oslo and the app will end up on the search page and you can see that we populate your search results for test, that value test is the value from the URL inserted into the application this is called reflected XSS (Cross Site Scripting), a very typical attack scenario.
If we find search features on websites, we can probably use them to cross site scripting, a very common vulnerability. So how do we exploit that? Here's the attack, we search for test, and we can also add an SVG image that we don't really display but when the SVG loads we want to insert the script tag into the page and load that attack.js file that's essentially a common way to attack an application.

Our malicious server is running, it's basically receiving that refresh-token. It also checks a heartbeat, so it detects that the application is alive because every couple of seconds we're like here's the token, the application is still alive, message is basically the check on the server side like every second it checks if the application is alive. What this now means is, if the malicious server knows the application is still using refresh-token rotation, there is no reuse of token. Everything works as expected until the application goes offline, once it has taken over that
refresh-token change from the application so the attacker now has access in your name, and they can do whatever they want.

Figure 46. Malicious software stolen refresh-token.
4.3 Solution Intercepting

In the above particular case the implementation of the proposed solution limits the malicious software not to go further beyond the time length of the access-token life span, the malicious software can never make use of refresh-token unless and otherwise. It can provide the one time password (OTP), which is almost unlikely.

The solution intercepting the flow is a huge plus to the flow of OAuth 2.0 authorization and the impact of having that in place is vital. Even in terms of user experience we are only adding a time not more than 5 seconds of task on the resource owner’s time. But the benefit, it is extraordinary. And one more thing, we are enhancing the security of REST API, which means we are adding an extra layer of protection for the very modern web architecture which makes human life much easier, and this is not something optional.

By design, we are tolerating, at least for now, only one time OTP error, if the one time password provided is doesn’t match with the one the STS reads from the authenticator, STS should revoke tokens and redirect client application to configured sign-out redirect call back page.

![Figure 47. Enhancement in action (access denying to malicious software).](image-url)
Figure 47. Section 1. Enhancement in action (access denying to malicious software).
Figure 47. Section 2. Enhancement in action (access denying to malicious software).
Figure 48. Enhancement in action (workflow).
CHAPTER 5

CONCLUSION

The protocol OAuth 2.0 is heavily used for authorization and access control in modern web and mobile applications. It’s a tight, safe, and flexible way to grant third-party applications access to the owner’s resources without exposing and sharing their credentials. Key points about OAuth 2.0 access control include:

- Authorization Workflow: OAuth 2.0 standardizes an authorization workflow involving the roles of the end user, client application, STS (Identity Provider or authorization server), REST API, and resource server. This workflow allows users to grant controlled access to their data.

- Access-tokens: OAuth 2.0 uses access-tokens, which are short-lived that grant permission to access specific resources. Access-tokens are issued by the STS (Secure Token Service) and presented to the REST API and to the resource server finally.

- Scopes: OAuth 2.0 introduces the concept of scopes, which define the specific permissions associated with an access-token. Clients request access to specific scopes, and users can grant or deny these permissions.

- Security: OAuth 2.0 provides a robust security structure of framework with various flows, such as the authorization code flow, hybrid, and the implicit flow. These flows help prevent unauthorized access and protect user data.

- Use Cases: OAuth 2.0 is commonly used for scenarios like social login, allowing users to sign into third-party applications using their existing credentials, and API access, enabling applications to access user data on their behalf.

- Implementations: There are many libraries and frameworks available for implementing OAuth 2.0, making it easier for developers to integrate this access control mechanism into their applications. In conclusion, OAuth 2.0 is a powerful and widely adopted framework for access control and authorization, ensuring the security and privacy of user data while
enabling seamless integration with third-party applications. However, proper implementation and adherence to best practices are crucial for maintaining the security and trust of the ecosystem.

5.1 Summary of Contribution

In this thesis, I focused on enhancing security in OAuth 2.0 by proposing the implementation of multi-factor authentication, specifically during the use of refresh-token. This involved integrating additional verification step beyond the initial authentication, ensuring a more robust and secure authorization process, and adding one more redirection and tweak of codebases in authorization and token end points, at least, at the STS level.

As one of the widely used framework, OAuth 2.0 is used to secure third-party access to resources, and as REST API is widely used in modern software and utilizes OAuth 2.0 highly, contribution to this security standard and protocol results in addressing vulnerabilities such as token leakage. And handling token leakage in a proper way before it causes a vast amount of damage is of huge importance. In this paper I tried to show how this can well be handled using multi factor authentication while using refresh-token in order to be granted an access-token for utilization and processing of REST APIs.

Making sure the request is coming from the legit client is the major goal here, and thus any kind of token leakage howsoever can be mitigated and can be stopped. There is no such thing as a fully secured system, but what’s important is protecting damage from happening and recovering as soon as possible.

5.2 Future Works

The proposed solution I proposed is based on the OAuth 2.0 protocol and can be implemented for any identity provider (STS) which follows the protocol, and even may be potentially other STSs with different protocols.

One of the major areas we can implement the proposed solution can be in application system where we have a single sign on feature. Federated sign-in is an authentication process where users are able to access multiple systems or applications with a single set of credentials. This
approach relies on identity federation, where a user's authentication information is shared securely across different trust domains. Federated sign-in promotes seamless and secure access to various services without the need for separate credentials for each application, enhancing user experience and simplifying identity management.

That's an interesting use case. You can't use a refresh-token anymore in that case because that doesn't involve resource owner interaction. What you can do is start a new authentication request (challenge the OIDC scheme again) when the current access-token is about to expire / has expired. That will redirect to the IDP. At level of that IDP, you'll still be authenticated because of the IDP's session cookie, so you won't have to provide username and password again. It's at that moment that you could force the user to enter MFA credentials again: at level of the IDP, when an existing session is still valid, and before providing new tokens.

What I have proposed to be implemented here in the solution is a really great resource for Federated Sign-In, without a client requiring re-authentication and only the resource owner providing a one-time password (OTP). User experience is altered very insignificantly, and security enhancement is boosted to a great value.

Lastly, based on findings of my research, it is highly likely that we can even omit the use of refresh token and just implement an observable and when the access-token is expired we an fire an event which redirects the web client page to a level of STS to show the one time password (OTP) prompt so that the client enters the value and if that checks out resource owner can continue working. The question here might be usability (poor user experience). So, this can simply be modified to show a dialog window without reloading the web client page.

This can be achieved with a request, starting at the authorize endpoint with a callback URL, that is the one time password (OTP) page. The parameter in the URL of prompt can be “otp”, which tells the STS to only proceed if it can complete the request if the OTP value provided checks out. Then it can immediately redirect back to the client’s otp-callback page, which potentially can be the page the resource owner was at. So now, as long as the resource owner is alive (is active) and the client’s session with the STS is alive, access tokens will always be valid after OTP checks out.
The key thing is to just ensure client side token expiration is less than the STS’s by some minimum margin.
REFERENCES


• OpenID Connect Core 1.0 Incorporating Errata Set 1. https://openid.net/specs/openid-connect-core-1_0.html


• OWASP Top 10:2021 https://owasp.org/Top10/A00_2021_Introduction/


BIOGRAPHICAL SKETCH

Ahadu Melesse, a native of Addis Ababa, Ethiopia, grew up in the city. Having finished his education at Catholic Cathedral School in Addis Ababa, he pursued higher studies at SMU in the same city, earning a Bachelor of Science in Computer Science in 2011. In the subsequent years, he gained professional experience in various IT companies, such as IBM, GuideWell, Delta Dental, to name some, where he worked as a Software Engineer. In August 2021, Ahadu enrolled in the Software Engineering graduate program at The University of Texas at Dallas (UTD).
Ahadu B. Melesse
Email: ahadu.melesse@yahoo.com

EDUCATION
Master of Science in Software Engineering (2023)
University of Texas at Dallas

Bachelor of Science in Computer Science
University of Saint Mary’s

PROFESSIONAL EXPERIENCE
Delta Dental (NEDD) 08/2018 – present
Software Engineer

GuideWell (AmWins) 08/2018 – 02/2021
Software Engineer

IBM 10/2015 – 08/2018
Software Engineer

ITG 01/2015 – 10/2015
Developer

RESEARCH INTERESTS
Security of REST APIs (ASP.NET Core)

TECHNICAL SKILLS
Development: C#, .Net Core, Angular2 (TypeScript), Angular Material, IdentityServer4, Entity Framework Core, Git (Source Control), Java, PowerShell, Java, C++, LAMP
Source Control: GIT, TFS
Web Technologies/Web Services: Angular, Angular Material, ASP.NET Core, TypeScript, CSS, HTML 5, Web Services (RESTful APIs), JavaScript, PHP, XSLT, ASP.NET
IDE: Visual Studio, VS Code, NetBeans, Eclipse
Frameworks: .Net Core, Angular, MVC, .Net 6, Greenfoot
**Database:** SQL Server 2019/2014/2012/2008, Oracle, MySQL, Mongo

**Operating System:** Windows 10/2008/2003, Linux, Mac OS X

**Applications & Software:** Adobe Photoshop, Visio