ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL

DESIGN AND DEVELOPMENT OF A SECURE AND ACCESSIBLE WEB AUTHENTICATION ALTERNATIVE TO FIDO2

M.Sc. THESIS

Ahmet DROBI

Applied Informatics

Cybersecurity Engineering and Cryptography Program

DECEMBER 2023

ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL

DESIGN AND DEVELOPMENT OF A SECURE AND ACCESSIBLE WEB AUTHENTICATION ALTERNATIVE TO FIDO2

M.Sc. THESIS

Ahmet DROBI
(707201014)

Applied Informatics

Cybersecurity Engineering and Cryptography Program

Thesis Advisor: Prof. Dr. Kemal BIÇAKCI

DECEMBER 2023

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ LİSANSÜSTÜ EĞİTİM ENSTİTÜSÜ

FIDO2'YE ALTERNATIF GÜVENLİ VE ERİŞİLEBİLİR BİR WEB KİMLİK DOĞRULAMA TASARIMI VE GELIŞTİRİLMESİ

YÜKSEK LİSANS TEZİ

Ahmet DROBI
(707201014)

Bilişim Uygulamaları

Bilgi Güvenligi Mühendisliği ve Kriptografi Programı

Tez Danışmanı: Prof. Dr. Kemal BIÇAKCI

ARALIK 2023

Ahmet DROBI, a M.Sc. student of ITU Graduate School student ID 707201014 successfully defended the thesis entitled “DESIGN AND DEVELOPMENT OF A SECURE AND ACCESSIBLE WEB AUTHENTICATION ALTERNATIVE TO FIDO2”, which he prepared after fulfilling the requirements specified in the associated legislations, before the jury whose signatures are below.

Thesis Advisor :	**Prof. Dr. Kemal BIÇAKCI** Istanbul Technical University	.....
Jury Members :	**Prof. Dr. Kemal BIÇAKCI** Istanbul Technical University	.....
	Prof. Dr. Enver ÖZDEMİR Istanbul Technical University	.....
	Asst. Prof. Mustafa Utku KALAY Yıldız Technical University	.....

Date of Submission : **29 December 2023**
Date of Defense : **6 February 2024**

To my greatest asset in the world - my family and friends

FOREWORD

Proposing an authentication protocol is never a simple task, especially when it's being offered as an alternative to a recent protocol supported by tech giants such as Google and Microsoft. Nonetheless, I hope that my proposal, flawed as it may be, can act as a guide on what FIDO's shortcomings are, and what can be done about them. It's my belief that for any password-based authentication flow to be widely adopted, it must be easily integrated into existing structures, and be easy for the average user to adopt. I hope that this belief is reflected in my work.

I would like to express my deepest gratitude to those who have supported me throughout working on this project. First and foremost, I thank my family for putting up with my constant complaints, and for believing in my ability to finish my master's. Second, my appreciation goes to my friends who at no point had doubts about my abilities and were always there when I needed an extra hand to help manage my time. I also extend my appreciation to the faculty and staff at the Informatics Institute of Istanbul Technical University for their guidance and mentorship, particularly Prof. Dr. Kemal Bıçakcı for his dedication to my academic growth and for pushing me to always be the best version of myself.

While reading my thesis, I hope that you will take the time not to just read the implementation details but to also understand what the aim of the proposal is - not just another authentication flow, but a push to move away from passwords while not compromising on what makes passwords great: ease of use, and ease of adaptability. This proposal is not but a drop in the greater ocean of scientific work trying to solve the password problem, yet my wish is that this thesis will be a valuable resource for others working on tackling this problem.

December 2023 Ahmet DROBI

TABLE OF CONTENTS

ABBREVIATIONS

QR	**:** Quick Response
FIDO	**:** Fast IDentity Online
2FA	**:** Two Factor Authentication
WebAuthn	**:** Web Authentication
CTAP	**:** Client to Authenticator Protocol
RP	**:** Relying Party
MITM	**:** Man-In-The-Middle
JS	**:** JavaScript
API	**:** Application Programming Interface
OIDC	**:** OpenID Connect
SCPC	**:** Savvy Code Delivering Focus
UFA	**:** Universal Authentication Framework
U2F	**:** Universal Second Factor
W3C	**:** World Wide Web Consortium
OAuth	**:** Open Authorization
PKCE	**:** Proof Key of Code Exchange
HTTPS	**:** Hypertext Transfer Protocol Secure
NFC	**:** Near-Field Communication
USB	**:** Universal Serial Bus
OS	**:** Operating System
URL	**:** Uniform Resource Locator
ID	**:** IDentification
RSA	**:** Rivest Shamir Adleman
HTML	**:** HyperText Markup Language
FA	**:** FIDO Alliance
PAKE	**:** Password Authenticated Key Exchange
IoT	**:** Internet of Things
PIN	**:** Personal Identification Number
REST	**:** Representational State Transfer

LIST OF TABLES

Page

LIST OF FIGURES

DESIGN AND DEVELOPMENT OF A SECURE AND ACCESSIBLE WEB AUTHENTICATION ALTERNATIVE TO FIDO2

SUMMARY

In embarking on this project, our primary goal was to introduce an innovative QR code-based authentication flow named QRAuth crafted to address the inherent limitations and challenges posed by traditional password-based and FIDO2-based authentication methods. Through a comprehensive analysis of FIDO2, delving into its workings and inherent requirements, we identified two pivotal pain points that we anticipate will pose significant hurdles to its widespread adoption. Firstly, the demand for "FIDO certified" devices running "FIDO certified" software that support WebAuthn and CTAP protocols. Secondly, the absence of account delegation support, a feature, albeit to a limited extent, is present in traditional passwords.

The essence of our proposal lies in not compromising the security of FIDO2 while adeptly resolving the aforementioned issues. Leveraging the same public-key cryptography as used in FIDO2, gives us a basis for the security of our protocol. By building on such a basis, we don't have to go over the "crypto" parts of our protocol and focus more on the application-level discussions. Readers interested in learning how exactly the underlying public-key cryptography works can read through the FIDO2 white paper [1]. However, as explored in our security analysis, our solution is by no means a perfect one and still has issues that need to be worked on, some of which are shared with FIDO2.

To overcome the shortcomings of FIDO2, our authentication flow transcends the conventional three components seen in FIDO2—RP, authenticator, and user agent (browser)—by introducing a pivotal browser extension component. Furthermore, we overhaul the other components to remove the need for "certification" and specific security protocols. Furthermore, our protocol is designed to ensure that all communication occurs over secure lines among the flow's four components[1].

The browser extension, an integral element of our system, assumes the crucial role of generating and displaying a QR code. This code encapsulates a challenge and other pertinent authentication information, such as the username. Our motivation for incorporating a browser extension stems from the need to satisfy the phishing-attack resistance property. The extension handles authenticating that the active site is genuine preventing MITM attacks. Our sample extension, developed using Chrome V3 API and entirely scripted in JS, ensures seamless portability to any other browser. Of noteworthy significance is that, as the user reads the QR code and completes the authentication process, the extension initiates an authentication-status polling request

[1]Except for the action of reading the QR code itself, which is a necessary by-product of using QR codes.

to the authentication server. Upon receiving confirmation of user authentication, the extension dynamically refreshes the user's web page, facilitating the reception of the authentication token.

Subsequently, utilizing the mobile authenticator, the user reads the QR code presented by the browser extension. The role of the mobile authenticator mirrors that of the authenticator in FIDO2, which involves the generation of public-private key pairs, signing the challenge using the private key, securely storing the private key in the device, and communicating the signed challenge with the public key to the authentication server.

Once the challenge is signed, it is transmitted to the authentication server alongside the user's public key (in login scenarios, only the signed challenge is transmitted). The server, after ensuring the user session hasn't expired, verifies the signature using the corresponding public key. Upon successful verification, the authentication server responds to the browser extension's polling request, confirming that the user is authenticated. Moreover, the authentication server issues an authentication token that allows the user to access protected resources.

A distinguishing feature of our flow is its support for temporary account delegation. This functionality empowers users to share QR codes with trusted entities, affording them temporary access to their accounts without compromising security.

To simplify the security analysis requirement for our proposal, we implemented our flow as part of OpenId Connect using Tokens. Thus, instead of delving into a comprehensive security analysis of the entire authentication protocol, our focus was on analyzing the components altered by our flow from the traditional password-based authentication schema used with OIDC.

In our security analysis, we validate that the inclusion of a browser extension indeed fulfills our prerequisite for resistance against phishing attacks. Furthermore, we assert that our security assumptions need not extend beyond the user's device into the relaying party. However, the support for account delegation through QR code sharing introduced a new vulnerability related to replay attacks, for which we propose a plausible solution using session timers. Finally, we assert that given that our security is anchored in public-key cryptography, any risk associated with digital signatures concurrently poses a risk to the security of our flow—a consideration shared with FIDO2.

To fully highlight what makes our proposal an attractive alternative to both traditional passwords, and FIDO2 we compared the three authentication options across eleven criteria outlining what makes QRAuth stand out.

We offer exhaustive implementation guidelines for each component of our proposal including which open-source tools we used. These guidelines encompass code samples serving as references for developers interested in replicating our proposal's implementation in their applications. The full code base of all components can be found in the accompanying CD to this thesis.

In conclusion, we firmly believe that our QR code-based authentication flow offers substantial advantages over FIDO2. Nevertheless, we identify potential areas for

future research and enhancements, including a comprehensive security analysis of our protocol and the persistent challenge of account recovery, mirroring the same issue in FIDO2.

FIDO2'YE ALTERNATIF GÜVENLİ VE ERİŞİLEBİLİR BİR WEB KİMLİK DOĞRULAMA TASARIMI VE GELIŞTİRİLMESİ

ÖZET

Bu projeye başlarken öncelikli hedefimiz, geleneksel şifre tabanlı ve FIDO2 tabanlı kimlik doğrulama yöntemlerinin doğasında var olan sınırlamaları ve zorlukları ele almak için hazırlanmış QRAuth adlı yenilikçi bir QR kodu tabanlı kimlik doğrulama akışı sunmaktı. FIDO2'nin kapsamlı bir analizini yaparak, işleyişini ve doğal gereksinimlerini inceleyerek, yaygın olarak benimsenmesinin önünde önemli engeller oluşturacağını tahmin ettiğimiz iki önemli sorun noktası belirledik. Birincisi, WebAuthn ve CTAP protokollerini destekleyen "FIDO sertifikalı" yazılımları çalıştıran "FIDO sertifikalı" cihazlara olan talep. İkinci olarak, geleneksel şifrelerde sınırlı da olsa bulunan bir özellik olan hesap yetkilendirme desteğinin bulunmaması.

Önerimizin özü, yukarıda belirtilen sorunları ustalıkla çözerken FIDO2'nin güvenliğinden ödün vermemekte yatmaktadır. FIDO2'de kullanılan aynı açık anahtarlı kriptografiden yararlanmak, protokolümüzün güvenliği için bize bir temel sağlar. Böyle bir temel üzerine inşa ettiğimiz için protokolümüzün "kripto" kısımlarının üzerinden geçmek zorunda kalmıyor ve daha çok uygulama düzeyindeki tartışmalara odaklanıyoruz. Temel açık anahtar kriptografisinin tam olarak nasıl çalıştığını öğrenmek isteyen okuyucular FIDO2 teknik incelemesini [1] okuyabilirler. Bununla birlikte, güvenlik analizimizde incelendiği üzere, çözümümüz hiçbir şekilde mükemmel değildir ve hala üzerinde çalışılması gereken ve bazıları FIDO2 ile paylaşılan sorunları vardır.

FIDO2'nin eksikliklerinin üstesinden gelmek için, kimlik doğrulama akışımız FIDO2'de görülen geleneksel üç bileşeni (RP, kimlik doğrulayıcı ve kullanıcı aracısı (tarayıcı)) çok önemli bir tarayıcı uzantısı bileşeni getirerek aşmaktadır. Ayrıca, "sertifikasyon" ve özel güvenlik protokollerine olan ihtiyacı ortadan kaldırmak için diğer bileşenleri elden geçiriyoruz. Dahası, protokolümüz tüm iletişimin akışın dört bileşeni arasında güvenli hatlar üzerinden gerçekleşmesini sağlamak üzere tasarlanmıştır[2].

Sistemimizin ayrılmaz bir unsuru olan tarayıcı uzantısı, QR kodu oluşturma ve görüntüleme gibi önemli bir rol üstlenmektedir. Bu kod, bir meydan okumayı ve kullanıcı adı gibi diğer ilgili kimlik doğrulama bilgilerini kapsar. Bir tarayıcı uzantısını dahil etme motivasyonumuz, oltalama saldırısı direnci özelliğini karşılama ihtiyacından kaynaklanmaktadır. Uzantı, aktif sitenin gerçek olduğunu doğrulayarak MITM saldırılarını önler. Chrome V3 API kullanılarak geliştirilen ve tamamen JS ile yazılan örnek uzantımız, diğer tüm tarayıcılara sorunsuz bir şekilde taşınabilirlik

[2]QR kodlarının kullanılmasının gerekli bir yan ürünü olan QR kodunun kendisinin okunması eylemi hariç.

sağlar. Kullanıcı QR kodunu okuyup kimlik doğrulama işlemini tamamladığında, uzantının kimlik doğrulama sunucusuna bir kimlik doğrulama durumu yoklama isteği başlatması kayda değer bir öneme sahiptir. Kullanıcı kimlik doğrulama onayını aldıktan sonra, uzantı kullanıcının web sayfasını dinamik olarak yenileyerek kimlik doğrulama belirtecinin alınmasını kolaylaştırır.

Daha sonra, kullanıcı mobil kimlik doğrulayıcıyı kullanarak tarayıcı uzantısı tarafından sunulan QR kodunu okur. Mobil kimlik doğrulayıcının rolü, FIDO2'deki kimlik doğrulayıcının rolünü yansıtır; bu rol genel-özel anahtar çiftlerinin oluşturulmasını, özel anahtar kullanılarak meydan okumanın imzalanmasını, özel anahtarın cihazda güvenli bir şekilde saklanmasını ve imzalanan meydan okumanın genel anahtarla birlikte kimlik doğrulama sunucusuna iletilmesini içerir.

Meydan okuma imzalandıktan sonra, kullanıcının açık anahtarıyla birlikte kimlik doğrulama sunucusuna iletilir (oturum açma senaryolarında yalnızca imzalı meydan okuma iletilir). Sunucu, kullanıcı oturumunun süresinin dolmadığından emin olduktan sonra, ilgili açık anahtarı kullanarak imzayı doğrular. Doğrulamanın başarılı olması üzerine kimlik doğrulama sunucusu, tarayıcı uzantısının yoklama isteğine yanıt vererek kullanıcının kimliğinin doğrulandığını onaylar. Ayrıca, kimlik doğrulama sunucusu, kullanıcının korunan kaynaklara erişmesine izin veren bir kimlik doğrulama belirteci yayınlar.

Akışımızın ayırt edici bir özelliği de geçici hesap delegasyonunu desteklemesidir. Bu işlevsellik, kullanıcıların QR kodlarını güvenilir varlıklarla paylaşmalarını sağlayarak, güvenlikten ödün vermeden hesaplarına geçici olarak erişmelerini sağlar.

Teklifimizin güvenlik analizi gereksinimini basitleştirmek için akışımızı OpenId Connect'in bir parçası olarak Token'lar kullanarak uyguladık. Bu nedenle, tüm kimlik doğrulama protokolünün kapsamlı bir güvenlik analizini yapmak yerine, OIDC ile kullanılan geleneksel parola tabanlı kimlik doğrulama şemasından akışımız tarafından değiştirilen bileşenleri analiz etmeye odaklandık.

Güvenlik analizimizde, bir tarayıcı eklentisinin dahil edilmesinin kimlik avı saldırılarına karşı direnç için ön koşulumuzu gerçekten yerine getirdiğini doğruluyoruz. Ayrıca, güvenlik varsayımlarımızın kullanıcının cihazının ötesine geçerek aktaran tarafa uzanması gerekmediğini iddia ediyoruz. Bununla birlikte, QR kodu paylaşımı yoluyla hesap yetkilendirmesi desteği, oturum zamanlayıcıları kullanarak makul bir çözüm önerdiğimiz yeniden oynatma saldırılarıyla ilgili yeni bir güvenlik açığı ortaya çıkarmıştır. Son olarak, güvenliğimizin açık anahtar kriptografisine dayandığını göz önünde bulundurarak, dijital imzalarla ilişkili herhangi bir riskin, akışımızın güvenliğine de eş zamanlı olarak risk oluşturduğunu iddia ediyoruz - bu durum FIDO2 ile de paylaşılan bir husustur.

Geleneksel şifreler ve FIDO2'ye kıyasla teklifimizin neden cazip bir alternatif olduğunu tam anlamıyla vurgulamak için, QRAuth'u öne çıkaran on bir kriteri belirleyerek üç kimlik doğrulama seçeneğini karşılaştırdık.

Teklifimizin her bir bileşeni için, kullandığımız açık kaynak araçlarını da içerecek şekilde, kapsamlı uygulama yönergeleri sunuyoruz. Bu yönergeler, teklifimizin uygulamasını kendi uygulamalarında tekrarlamak isteyen geliştiricilere referans olacak

kaynak kod kapsamaktadır. Tüm bileşenlerin tam kod tabanı, bu teze eşlik eden CD'de bulunabilir.

Sonuç olarak, QR kod tabanlı kimlik doğrulama akışımızın, FIDO2'ye kıyasla önemli avantajlar sunduğuna kesinlikle inanıyoruz. Bununla birlikte, protokolümüzün kapsamlı bir güvenlik analizi ve FIDO2'deki aynı sorunu yansıtan hesap kurtarma konusundaki kalıcı zorluk da dahil olmak üzere gelecekteki araştırma ve geliştirmeler için potansiyel alanları belirliyoruz.

1. INTRODUCTION

The FIDO (Fast IDentity Online) alliance proposed authentication protocols that can act as either second-factor authentication methods (2FA) or complete substitutes for passwords. The protocol is comprised of three frameworks: The Universal Authentication Framework (UFA) that can be used for password-less authentication for smart devices; the Universal Second Factor protocol (U2F) can be used for two-factor authentication by using a token installed to a non-FIDO supported smart device having interacted with a FIDO-compliant website; and FIDO2 which was added into the W3C Web Authentication Recommendation. The FIDO2 protocol offers strong cryptographic security guarantees due in large part to its usage of public key cryptography. However, like any other protocol or specification trying to replace passwords, either partially or completely, FIDO2 faces the issue of widespread adoption and support [2]. This can be mainly attributed to the constraints put by FIDO2 on becoming a ”FIDO2 Certified” authenticator or device [3]. The adoption can be carried out in two steps: end-user adoption, and existing authentication protocol interactions. End-user adoption requires having FIDO-compliant devices (Smartphones, security keys, laptops, etc.) and transitioning from the currently intuitive password world. Secondly, the integration of FIDO2 into applications that have relied heavily on passwords for their authentication in the past is an essential step in the journey to completely replace passwords. We believe FIDO2's main issue will be possible reluctance among end-users due to numerous reasons including cost factors and usability hurdles [4].

1.1 Purpose of Thesis

Using the shortcomings of FIDO2 as our guide, we propose in this thesis a novel QR-based authentication flow, named QRAuth, that uses a browser extension to provide phishing resistance for QR Code-based authentication flows [5]. As a

cornerstone of our project, we want to build on FIDO2 by providing a public-key cryptography-based authentication flow that solves the accessibility and account delegation issues of FIDO2. As a metric of our success, we aim to create a security analysis of our protocol as well as comparing our solution to both FIDO2 and classical password authentication flows as a way to prove how our protocol builds on the security of FIDO2 while maintaining accessibility and simplicity in integration.

1.2 Literature Review

The quest to replace passwords has been an active area of research for a long period. As such, to not repeat other researchers' work on why the need to replace passwords exists, and how to evaluate new alternatives we will refer to the work we used while working on our project.

Han et al. [6] survey on different attack vectors against passwords highlight the inherent vulnerability of passwords concerning human errors. While such human errors can be reduced using relatable data techniques as shown by Carstens et al. [7] their survey shows that even in best-case scenarios where users do not lose track of their text-based passwords, their ability to keep it safe, as in not sharing it with an attacker using a common phishing attack technique [8] is limited and can't be relied upon for the security of a system.

The analysis originally done by Bonneau et al. [9], then revisited and updated by Zimmermann et al. [10] on past text-based password alternatives gave us a guide on the pitfalls that our proposal should avoid, and using the benchmark developed by their work, we were able to better design our proposal to be user-friendly aiming to pass the benchmarks set in both papers.

User's openness to adopt alternatives to text-based passwords is evidently of vital interest to us, especially where the proposed solution uses automated key generation (i.e: asymmetric key) instead of a master password as proposed by Fukumitsu et al. [11]. Consequently, Lyastani et al. [12] comparative usability study of FIDO2 is of great interest to our work. Their results show that users are open to accepting a complete replacement of text-based passwords with a security key as a single-factor

authenticator. Furthermore, their work identified five gaps in FIDO2 that could hinder its ability to replace passwords. Of interest to us, are the two points regarding the lack of authenticator revocation, and ease of usability, that we tried to overcome in our proposal.

FIDO2 is not the first authentication proposal from FA. As such, tracking the various changes that led from the initial creation of FIDO to the current protocol we examine in our paper, can give us an idea of the future direction that FA is taking the FIDO protocol in. We used the survey done by Angelogianni et al. [13] for this purpose.

As a promising recent alternative to passwords, extensive research has been done in recent years to analyze FIDO2's security. The analysis done by Barbosa et al. [14] is unique in their approach to analyzing FIDO2's WebAuth and CTAP components. In their work, they confirm the security of WebAuthn alongside the security guarantees of the underlying public-key cryptography used in the protocol, however, since CTAP is based on unauthenticated Diffie-Hellman, it fails their security model.

Guan et al. [15] show that FIDO2 still fails in some authentication and usability properties such as account recovery, and FIDO's solution of using PINs lessens the protocol's security level.

Using QR codes as part of an authentication system has been proposed before in other authentication mediums such as the protocol developed by Oh et al. [16] for mobile cloud computing. We took the inspiration for the generation of the QR code from their proposal. Another novel usage of QR codes in an authentication schema is the work done by Al-Ghaili et al. [17] to implement an IoT authentication method that leverages a multi-security layer of rotating QR codes to establish secure authentication for users.

Our work is similar to the interesting protocol proposed by Preetham et al. [18] which relies on SCPC coupled with changing PIN numbers. Our proposal removes the need for SCPC by relying on a single authentication provider. Kim and Jun [19] proposed a different method for generating QR codes and subsequent user authentication. Their proposal is based on a QR code calculated from a collection of device-specific information like the serial number XOR'ed with a predefined user PIN value. The

way QR code gets generated in their work makes it open to re-transmission attacks, as indicated in their security analysis.

The work done by Siwon et al. [20] for the analysis of the security of a proposed QR code authentication method inspired us for the security analysis of our solution. Chengqian [21] proposal of extending FIDO2 to use QR-codes to authenticate user agents helps resolve some of the issues faced by FIDO2 and is comparable to our proposal. However, as their work is integrated as part of FIDO2 its actual adaption is not possible until the approval of the FIDO Alliance, something as far as we researched has not happened yet.

As our proposal relies heavily on QR codes, it's imperative to understand the security of QR code usage in general even outside authentication flows. Krombholz et al. [22] conducted a security analysis that focuses on the apps that use QR codes' weakness to phishing attacks. Since users can't easily verify the authenticity of a QR code before scanning it, malicious attackers can replace genuine codes with malicious codes that can be used to trick users into providing their secure information to the attacker. This weakness, as explored by Vidas et al. [23], was firmly in our mind when we designed our proposal and is the motivation behind our inclusion of a trusted browser extension.

Another security side of QR codes is the QR code scanner that the users will use. Our proposal includes a mobile app that handles the reading of the QR code and ensures that the keys are saved safely. As such, users of our proposal will need to completely trust that the mobile authenticator is safe and to be trusted. All of this means, that the security of the QR code scanner - which is part of the authenticator in our proposal - is of great importance. Dudheria [24] evaluated what such security should be, and we used his analysis while developing our scanner.

The work done by Stefan [25] to encode parts of the OIDC request into QR codes to make OIDC more widely adaptable is of interest to us since it includes using QR codes in OIDC. However, his work is limited to encoding the OIDC request itself into a QR code instead of using a QR code in the user authentication steps as we do in our proposal. This different usage of QR codes within OIDC opens the door to exploring if

the whole protocol can be managed through communications using QR codes instead of REST redirections.

Finally, while our security analysis focus is on the parts of OIDC that QRAuth changes, it's important to understand the limits of the security of the other components in OIDC and how secure the overall protocol is. For this, we found the work done by Fett et al. [26], Navas and Beltrán [27] and Minka et al. [28] extremely informative and helped us recognize what aspects of OIDC can be modified without compromising the security of the protocol as a whole.

2. FIDO2 AUTHENTICATION PROTOCOL

The FIDO alliance proposed an authentication protocol that can act as either a 2FA authentication method or a complete substitute for passwords. The protocol is comprised of three frameworks: The Universal Authentication Framework which can be used for password-less authentication for smart devices; the Universal Second Factor protocol which can be used for two-factor authentication by using a token that's installed to a non-FIDO-supported smart device having interacted with a FIDO compliant website; and FIDO2 which was added into the W3C Web Authentication Recommendation. The FIDO2 protocol offers strong cryptographic security guarantees due in large part to its usage of public key cryptography. The protocol provides specifications for registration and subsequent authentication without any reliance on users maintaining passwords. However, like any other protocol or specification trying to replace passwords, either partially or completely, FIDO2 faces the issue of widespread adaptation and support. The protocol adaptation can be categorized into two parts: end-users' adaptation, and existing authentication protocols integration. For end-users adaptation means having FIDO-compliant devices (Smartphones, security keys, laptops, etc.) and transitioning from the currently intuitive password world. Moreover, the integration of FIDO2 into popular and widely used authentication protocols that have relied heavily on passwords for their authentication in the past is an essential step in the protocol's journey to completely replace passwords. As part of this project, we first worked on the second point by integrating FIDO2 into OAtuh PKCE flow. The OAuth PKCE flow is an OAuth flow that builds on the popular standard Authorization Code Flow to help mitigate security vulnerabilities that come from public clients (such as native and single-page applications) inability to securely store secrets. It relies almost completely on its security on a password-based end-user authentication step.

2.1 OAuth PKCE Authentication Flow

What separates the PKCE flow from the standard Authorization Code Flow is that it introduces a secret that gets generated by the client (i.e.: the calling application) that gets verified by the authorization server. This code is named the Code Verifier. Furthermore, the client also generates a transformed value from the Code Verifier called the Code Challenge. The code challenge subsequently is sent over HTTPS to get an Authorization Code. The main advantage of this flow is that an attacker can only intercept the Authorization Code, and they cannot exchange it for an access token without the original Code Verifier.

2.1.1 The protocol flow

Figure 2.1 shows clearly how the protocol works. The flow's steps are:

1. The user first initializes the flow by clicking on a login link.
2. Then the app generates a cryptographically random Code Verifier and from it generates a code challenge.
3. The code challenge is then sent to the authorization server as part of an Authorization Code Request.
4. The authorization server redirects the user to a login page.
5. The user authenticates themselves using an account created beforehand.
6. Upon successful user authentication, the authorization server stores the Code Challenge and redirects the user back to the client with a one-time-use authorization code.
7. When the client wants to access a protected service/API, it sends the authorization code received in step 6 with the code verifier to the authorization server in order to receive an authentication token.
8. The authorization server verifies the Code Challenge and the Code Verifier.
9. The authorization server sends an access token back to the client.

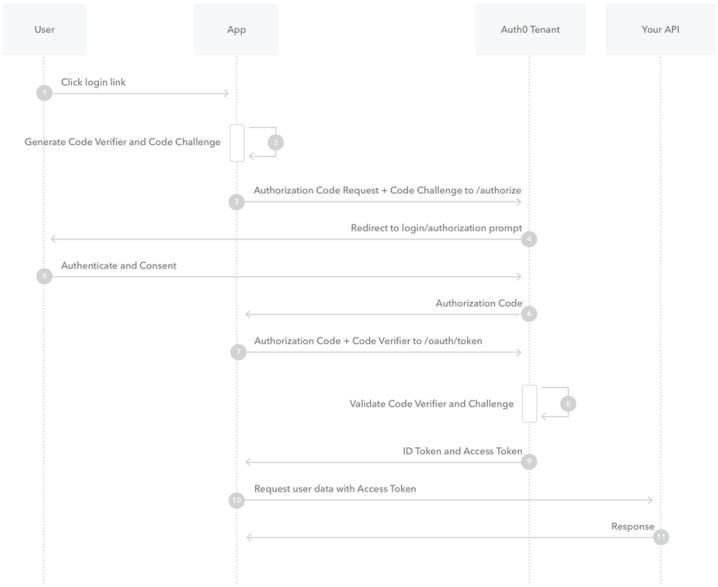

Figure 2.1 : PKCE Flow.

10. The client uses the access token to access the protected services.

2.2 FIDO2

FIDO2 is an authentication protocol that's built on public-key cryptography and is aimed at replacing or at least reinforcing password-based authentication as a 2FA. The protocol is based on 3 main components:

- A relying party is the server that the user wishes to authenticate to.
- JavaScript application which is the website the user is accessing running in the browser (could also be a native app). Any operations that happen in the JS application are conducted "offline" within the user's environment unless otherwise stated.
- Authenticator is the user's device that will generate public-private key pairs, handle encryption of challenges, and store the user's private key. The authenticator is usually the user's mobile but could also be a security key or third-party device that supports the FIDO2 protocol as an authenticator.

Optionally, the relying party that both acts as an authentication server and "data" provider can be further divided into two components:

- An authentication server that handles the creation of challenge, storing the user's public key and other authentication-related operations.
- A service or data server that only verifies that the user is authenticated by checking a token generated by the authentication server is valid (usually by contacting the authentication server) and provides the protected data.

This separation is recommended since it would allow the data server to be more flexible in how it can integrate with different authentication mechanisms and also shifts the burden of setting up a secure and correct implementation of the FIDO2 authentication flow to a contained server. To facilitate communication between a website or native app running in a browser and the user's device FIDO2 relies on WebAuthn which is a web standard published by the W3C. WebAuthn provides a standardized interface for authenticating users on web-based applications using public-key cryptography. The device that acts as an authenticator must confirm to CTAP which is part of an older version of the FIDO protocol. What makes WebAuthn and CTAP crucial for the FIDO2 protocol is that it allows secure communication between browsers and roaming devices using Bluetooth, USB, or NFC without sending data over the internet ensuring that communication is resistant to phishing attacks. The usage of CTAP in FIDO2 ensures that hardware authenticators are safe from malware since the private key material is at no time accessible to software running on the host machine. The flow of the FIDO2 protocol differs based on which part of it is to be used, for this project we implemented the password-less authentication flow which works as follows for registration:

1. The user navigates to a website or native app that supports WebAuth and makes an authentication request.
2. The authentication server (part of the relaying party) sends back a response that contains the parameters that the authentication client (authenticator) must fulfill to register.
3. The WebAuthn clients get the object returned by the authenticators and check the availability of authenticators that are connected. An important note here is that the authentication server is assigned an ID based on the domain name and it must

be running HTTPS over TLS. Consequently, the authentication server ID is only valid if it matches the origin of the authentication server. For example, assume that the URL for login is *https://www.example.com/login* the following list of origins is valid: login.example.com or example.com but origins like m.login.example.com or www.example.net/login are invalid.

4. The authenticator then authenticates the user if required. This could be the case on a mobile phone using biometrics if supported or a simple PIN code used to protect the device.

5. After authenticating the user, the authenticator generates an asymmetric public-private key pair. The generated public key with information from the authentication server is signed with the authenticator's private key (called Attestation Private Key) and is sent back to the authentication server and stored for later authentication. The authenticator attestation is required so that the authenticator proves its authenticity. Optionally a signature counter can also be run after registration and is sent to the authentication server to be used to identify and reject impersonating authenticators.

6. The WebAuthn client receives the updated public key that was created and signed in the previous stage as well as any further attestations that were required. A certificate or other comparable data that can verify the public key's provenance information by the authentication server is typically included in the attestation statement along with a public key that has been digitally signed by the attestation private key and a challenge. The term "Attestation Object" is also used to describe this bundled final object.

7. The WebAuthn client then forwards the attestation object to the authentication server for verification.

8. If the authentication server verifies the attestation object it stores the user's public key which can be used in the future for authentication.

The flow for authentication is very similar and goes as follows:

1. The user navigates to a website or native app that supports WebAuth and makes an authentication request.

2. The authentication server sends back a response that contains a challenge and other information. The extra information sent back by the authentication server can be used to narrow down the options for the user to authenticate themselves to use the same device used in registration. Such information could be a username that was requested during the registration process.

3. The WebAuthn client, upon receiving the response from step 2, starts the validation process by the authenticator. The client would forward the challenge received from the authorization server with its ID and origin which should be the same as the ones used during registration.

4. When the authentication server ID from the registration matches what the authenticator has already saved, the authenticator creates an assertion signature. The generated private key of the authenticator during the registration process is used to authenticate and sign the data from the WebAuthn client. It might be necessary to complete local authentication on the Authenticator once more before this can be done.

5. The authenticator sends back the signed data to the WebAuthn client.

6. The WebAuthn client forwards it to the authentication server.

7. The authentication server checks that it has the public for the user and verifies the assertion signature.

2.3 Our Implementation

To fully showcase the complete synergy between the two protocols we have implemented the two possible usages of PKCE OAuth:

- For web-based sites APIs

- For native apps that can't store secrets

Both implementations share the same basic architecture which you can find in Figure 2.2. We created an authentication server using IdentityServer4 that we modified to support both PKCE and FIDO2, a basic data server that just returns weather information behind a protected endpoint and two WebAuthn clients: a React Native mobile app, and a JavaScript-enabled website. As an authenticator we used the available out-of-the-box Android authenticator that's part of the Android OS, we also tested using IOS.

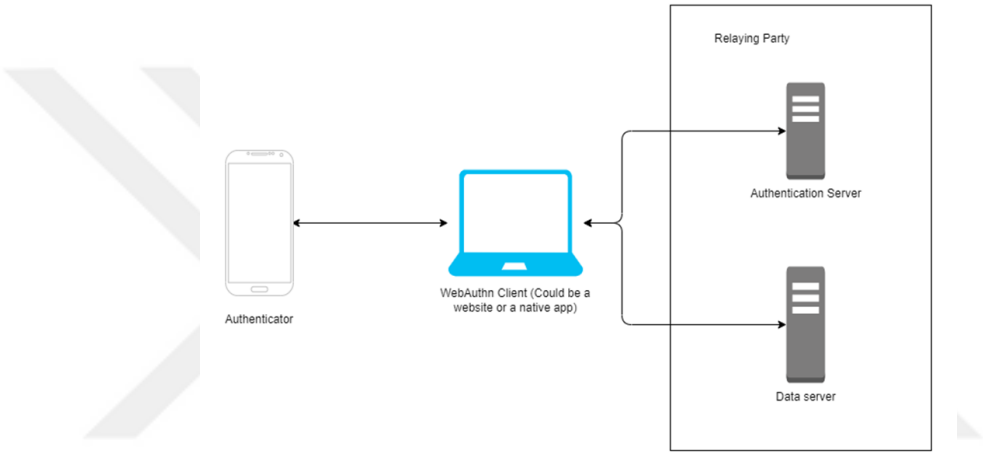

Figure 2.2 : Our FIDO2 system design diagram.

2.3.1 Authentication flow

To better understand how the overall integration we created works, the underlying FIDO2 authentication implementation must first be examined. Our FIDO2 registration flow implementation can be seen in Figure 2.3. For the sake of brevity, we didn't include the authentication flow since it's practically identical, however, it will be clear when the overall authentication flow is demonstrated.

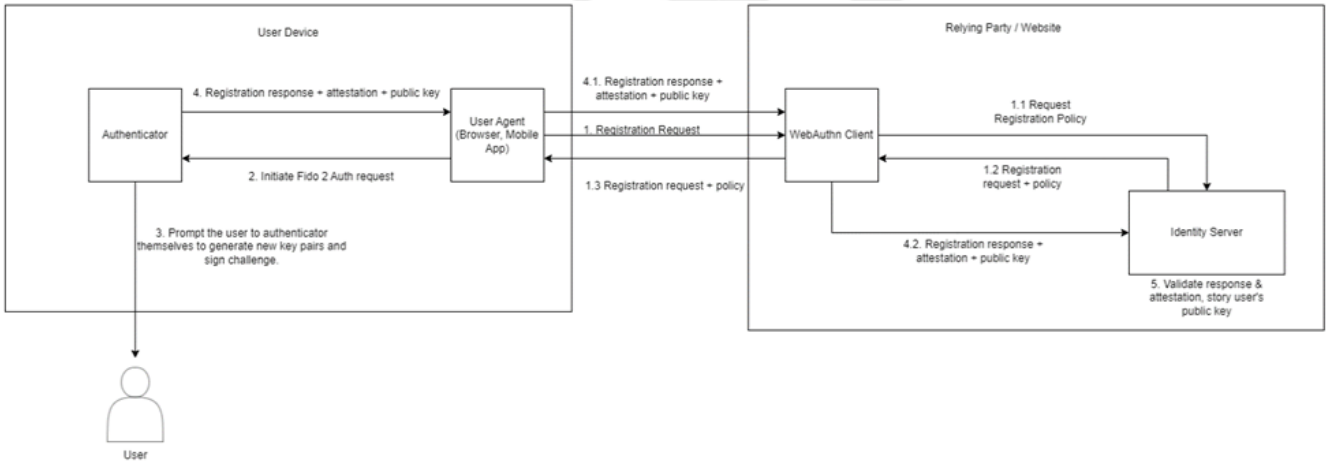

Figure 2.3 : Our FIDO2 registration implementation.

As can be clearly seen, our implementation follows the FIDO2 official guideline for implementation of the registration flow with just a minor implementation details difference. In our project, we used IdentityServer4 which is a popular identity and authentication server that's implemented using .Net. The open-source framework provided us with a bedrock of basic components needed for any authentication flow like session and cookie management, routing setup, etc... The framework itself doesn't have FIDO2 authentication endpoints or logic. So, all the necessary setup was created by us. The modifications made to Identityserver involved adding logic to support the following:

- Getting a valid FIDO2 registration policy that can be sent to authenticators.
- Getting a valid FIDO2 authentication policy that can be sent to authenticators.
- Validating responses from registration requests creating the user in the database and associating them with their public key.
- Validating responses from authentication requests returning an OAuth Authorization Code that can be then exchanged for an Authorization Token.

Note that validating the authorization code sent by the user, generating an Authorization Token, and finally validating the Authorization token was not built by us. We relied on the IdentityServer4 implementation of that part of the flow since it's the standard OAuth flow and has nothing interesting for our project to tinker with.

The code to achieve what was listed above is quite involved, however, we added it here in case you would like to examine it. Finally, for our FIDO2 authentication to function the Web App must also be FIDO2 compliant and be able to use WebAuthn protocol to communicate with the Authenticator and do all the related needed operations to support registration and authentication. Unfortunately, the code for that is also quite involved, however, to showcase it we took two screenshots of the login and registration code.

2.4 Combining OAuth Implementation with FIDO2

The complete sequence diagram of how the two protocols will now flow in our new implementation can be seen in Figure 2.4. The core of the implementation is swapping the normal password-based authentication that can be seen in step 5 in Figure 2.1 with our FIDO2 implementation. Figure 2.4 shows the authentication flow and not the registration one. The registration is done completely independently of OAuth PKCE flow and is done based on FIDO2 as can be seen in Figure 2.3

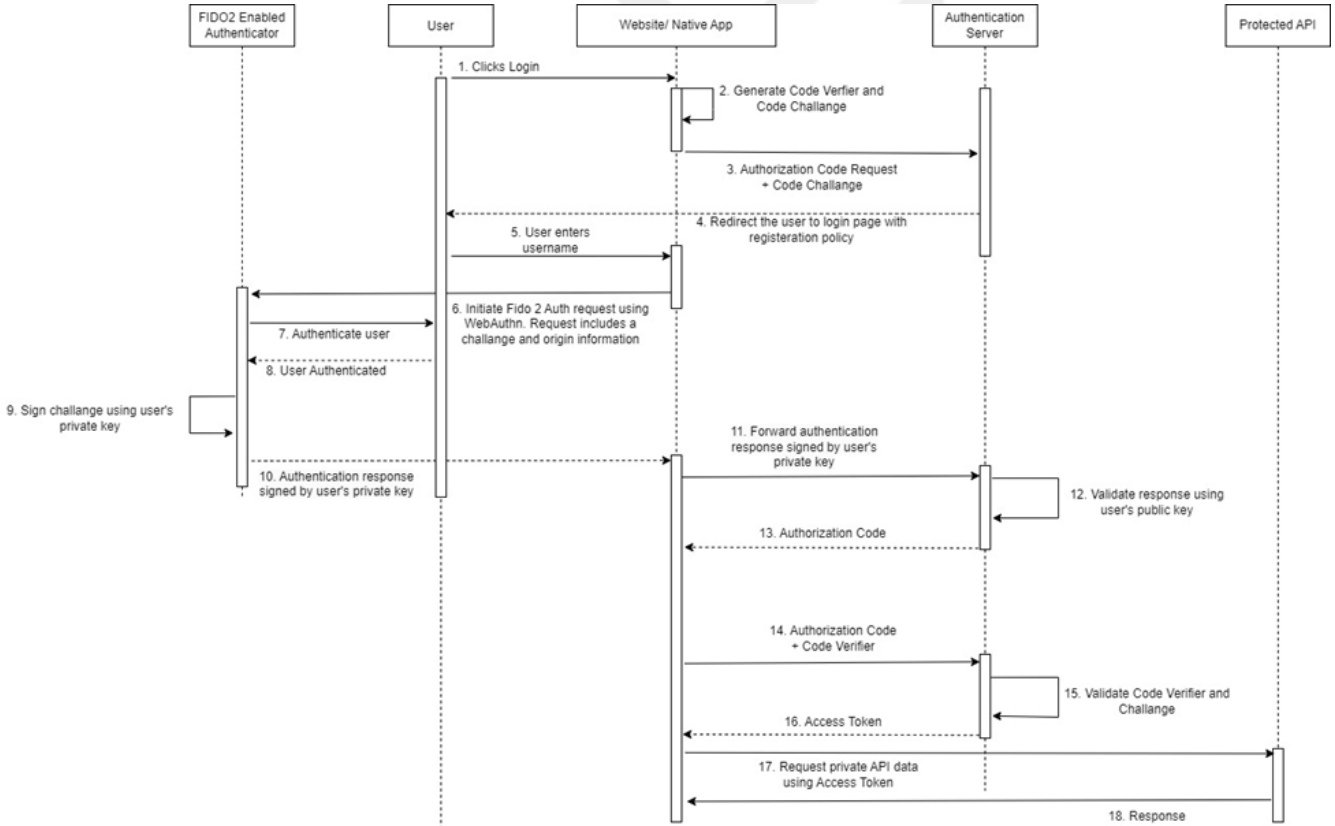

Figure 2.4 : Sequence diagram showing OAuth PKCE flow.

2.5 Native App Implementation

The main usage of PKCE flow is native apps that can't support API secrets keys safely, and so to demonstrate that our combined protocol is secure and fulfills all the usages of PKCE it must also work seamlessly with a native app. We put a lot of emphasis on seamless integration since the Authentication Server in our implementation operates independently of the user's device. That is, it shouldn't matter for the authentication server from which device or app the user is trying to authenticate or register themselves given that the means they are using has a valid implementation of both FIDO2 and PKCE flows. To achieve this, we created a React Native app that will work on both IOS and Android devices proving the cross-operating system compatibility of our implementation.

2.6 Outstanding Issues

Although we showed the interoperability of FIDO2 with the work we did, the first part of the issues facing FIDO2 still stand. Namely, the issue of adaptability. For end-users to authenticate themselves on a website using FIDO2 they must possess a FIDO2-certified device, running a FIDO2-certified authenticator. For most users, this will be an Android or IOS device running a newer OS version. The second requirement, however, is the issue. The vast majority of FIDO2-based authentication will happen using either Android or IOS devices and will be using Google's and Apple's built-in FIDO authenticators. This is a byproduct of the costly and very restrictive requirements for an authenticator to be certified by the FIDO Alliance. Consequently, end-users will have to put their trust in their device's manufacturer using their closed authenticators. Furthermore, any site wanting more control over how their user's secrets are kept has to be contented with the OS built-in support. This, we believe, not only goes against the fundamental idea behind public key cryptography where owners of the keys have complete control over their keys, it introduces a clear vulnerability point where a successful attack against either company's authenticator can have disastrous consequences. The second issue we see in FIDO2 is the lack of

support for authentication delegation. For normal passwords, a user can easily delegate access to their account to a subordinate for example. In a perfect world, the user would then revoke that access by changing the password. Although that is not a perfect solution for delegated access, wherein not only is the ease of process is important but also it should be temporary to avoid security vulnerability, it is still an option FIDO2 does not support at all

3. QR CODE BASED AUTHENTICATION FLOW

To solve the aforementioned issues, we created a new QR code-based authentication flow that we named QRAuth. Before explaining how the flow itself works, we want to highlight first the main components of it, and what makes it stand out from other authentication flows. First, authentication happens by communication from the user's device to the authentication server directly. This was done to protect against tempering in authentication messages, or MITM attacks. Second, we introduce a new component to authentication flows which is a browser extension. We created a sample extension that works with the Chrome browser, but the logic needed in the extension is usable in any browser. The benefit of this point will become clear when we explain how the flow works. Lastly, like FIDO2, our flow is based on public-key cryptography thus providing us with the same cryptographic security as other public-key-based authentication flows. To reduce the security analysis needed for the flow we opted to implement it as part of OpenID Connect using Tokens authentication protocol. So that when we do the security analysis, our only focus would be the new parts we propose instead of having to do a full analysis covering a full protocol.

3.1 OpenID Connect

OpenID Connect enables users to authenticate themselves to multiple websites or applications (relying parties) using a single set of credentials provided by an identity provider. It simplifies the user experience by eliminating the need to remember and manage multiple username-password combinations across various platforms. Instead, users can authenticate themselves to an identity provider, which then issues tokens that can be used for subsequent authentication and authorization with relying parties. Our implementation uses the "Token" version of the protocol that's called OIDC. Tokens are lightweight, digitally signed (by the identity server) pieces of information that contain specific claims about the authenticated user. These claims include user

identification information: a unique identifier, and an email address. Similar to OAuth, OpenID also relies on access tokens. For the purposes of this project, they function practically the same as they do in OAuth.

3.1.1 OpenID Connect authentication flow

To better illustrate the modification we made to OIDC in integrating our proposal it's necessary to go over how the "normal" or "unmodified" version of OIDC would look like. And so as can be seen in Figure 3.1. The steps of which are as follows:

1. The authentication flow starts when the user attempts to access a protected resource.
2. The user is redirected to a login page hosted by the identity server with some extra parameters attached such as a callback URL, and the client ID.
3. The user enters their username and password to authenticate.
4. The identity server checks the user information and authenticates them.
5. The identity server redirects the user to the relaying party callback attaching an authentication token (cookie).
6. The relying party verifies the authentication token received by using the authentication server's public key.
7. The relying party allows the user access to the protected resource.

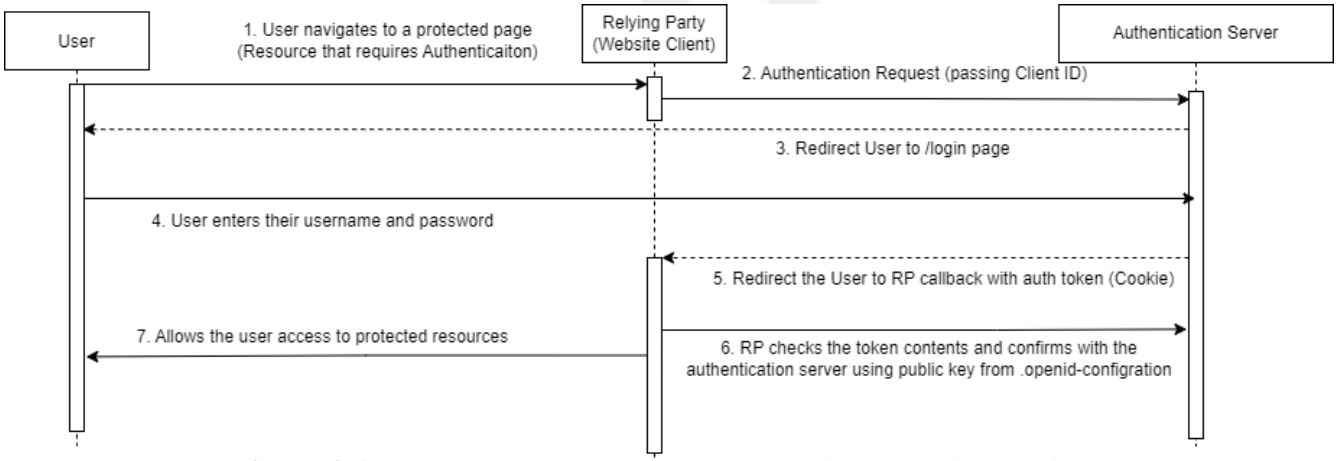

Figure 3.1 : Unmodified OpenID Connect login authentication flow.

The registration flow is very similar and can be seen in Figure 3.2. The steps of which are as follows:

- The authentication flow starts when the user attempts to access a protected resource.
- The user is redirected to an authentication page hosted by the identity server with some extra parameters attached such as a callback URL, and the client ID.
- The user signs up by entering their username and preferred password.
- identity server checks the user information and authenticates them.
- identity server redirects the user to the relaying party callback attaching an authentication token (cookie).
- The relying party verifies the authentication token received by using the authentication server's public key.
- The relaying party allows the user access to the protected resource.

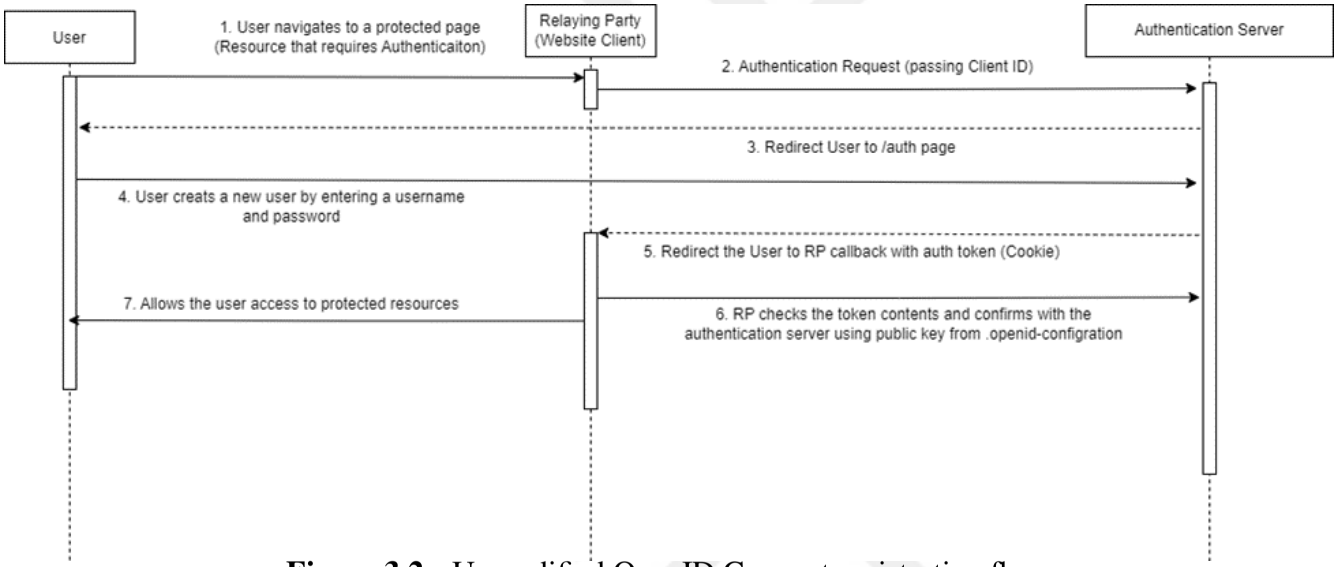

Figure 3.2 : Unmodified OpenID Connect registration flow.

3.2 QR-Code Based Authentication Flow

To solve the accessibility, trust, and account delegation issues of FIDO2 we propose the usage of QR codes. QR codes are bar-code images that usually represent a URL or encoded information. Other work has been done in the past to use QR codes as part of two-factor authentication flows, or as part of larger authentication protocols. A popular usage that might be familiar to users is WhatsApp web client. The web client prompts the user to scan a QR code using the WhatsApp mobile app that has their account active. In the background, the mobile app would send an authorization confirmation to the server that pings the web session to indicate that the user is authenticated. Our work intends to fully expand this concept to not be limited to a specific app or server, but to work with authentication servers and relying parties to fully support OpenId Connect while being deployable and integratable to existing web applications. Furthermore, to combat the issue of phishing attacks we move the burden of authentication from the website itself to a separate, website generic extension that handles verifying that the site is genuine and handles the communication with the authentication server. Such usage of a browser extension, as shown in the literature review is quite novel. As a cornerstone of our project, all secure communication must flow through secure mediums. That is, we only presuppose the security and integrity of the user's own mobile device, not any of the medium over which the communication happens. This is of great importance, as communication in our protocol flows generally between four anchor points:

- Authentication server.
- Mobile app (the authenticator).
- Relying party/Web app/Client.
- Browser extension.

The extension communicates with the authentication server to prepare the authentication request (The QR code is shown to the user), while the mobile app

reads the QR code and communicates directly with the authentication server (this is done to overcome the one-way nature of QR codes[1]) sending the required information for authentication such as a signed challenge or the user's public key. Finally, the authentication server in turn communicates back the authentication status to the relying party completing the authentication flow.

This method of triangular communication ensures that the only communication happening in insecure environments is the reading of the QR code. We overcome this necessity by ensuring that only the user's mobile app is capable of authenticating through the shown QR code. We will expand on this in the security analysis section.

First, to fully understand the flow, we will examine it on its own as if deployed as a normal atomic authentication flow. Starting with the registration flow, which can be seen in Figure 3.3, and goes as follows:

1. The user starts the registration by entering their username and clicking register.
2. The extension sends a request to the authentication server asking for a challenge to show the user.
3. The authentication server responds with the challenge and a session ID that's used in subsequent communication to track the active user session.
4. The extension starts an authentication status-pulling job that runs every half second.
5. The extension performs security checks such as checking that the domain of the authentication server for which the challenge was received matches the current site's domain to protect against phishing attacks.
6. The extension renders a QR code that contains the challenge and session ID received from the authentication server; the site domain, and the requested action type – in the case of registration it's modeled with a simple letter 'R'.
7. The user using his mobile phone reads the QR code.

[1] In theory, QR codes could also be used to have a bidirectional communication channel. For instance, a webcam could be used to read a QR code displayed on a mobile device. Since this is not a common use case for QR codes, we do not explore it further in this project

8. The mobile app then parses the QR code retrieving the domain.

9. The mobile app checks that no key pair exists for the domain and generates a new public-private key pair saving the private key in the secure storage of the device.

10. Using the private key, the mobile app signs the challenge.

11. The mobile app forwards the public key, the signed challenge, and the username to the authentication server directly.

12. The authentication server does the security checks such that the active session hasn't expired yet and verifies the signed challenge.

13. The authentication servers respond to the authentication status pulling job indicating that the user is authenticated.

14. The extension redirects the user to the site callback. When the page is loaded the user receives the authentication cookie with which they can access protected resources.

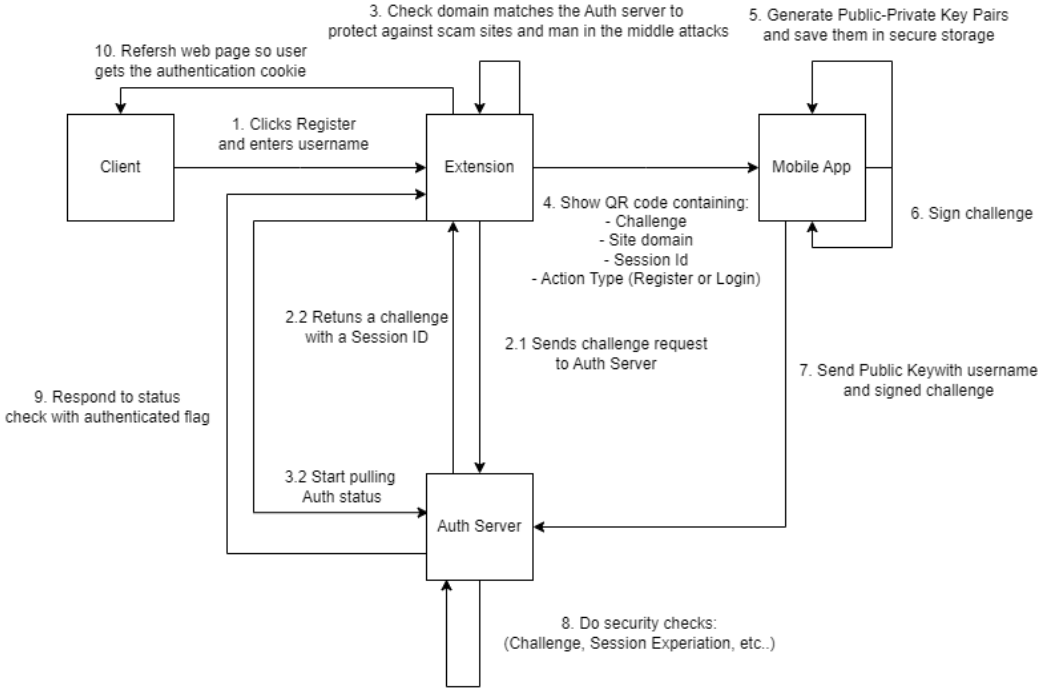

Figure 3.3 : QRAuth registration flow.

The login flow on the other hand is very similar with small differences as can be seen in Figure 3.4. The steps of it are:

1. The user logs in by entering their username and clicking login.
2. The extension sends a request to the authentication server asking for a challenge to show the user.
3. The authentication server responds with the challenge and a session ID that's used in subsequent communication to track the active user session.
4. The extension starts an authentication status-pulling job that runs every half second.
5. The extension performs security checks such as checking that the domain of the authentication server for which the challenge was received matches the current site's domain to protect against phishing attacks.
6. The extension renders a QR code that contains the challenge and session ID received from the authentication server; the site domain, and the requested action type – in the case of registration it's modeled with a simple letter 'L'.
7. The user using his mobile phone reads the QR code.
8. The mobile app then parses the QR code retrieving the domain.
9. The mobile app checks that a key pair exists for the domain and retrieves the private key associated with it.
10. Using the private key, the mobile app signs the challenge.
11. The mobile app forwards the signed challenge and the username to the authentication server directly.
12. The authentication server does the security checks such that the active session hasn't expired yet and verifies the signed challenge using the existing public key for the user.
13. The authentication servers respond to the authentication status pulling job indicating that the user is authenticated.

14. The extension redirects the user to the site callback. When the page is loaded the user receives the authentication cookie with which they can access protected resources.

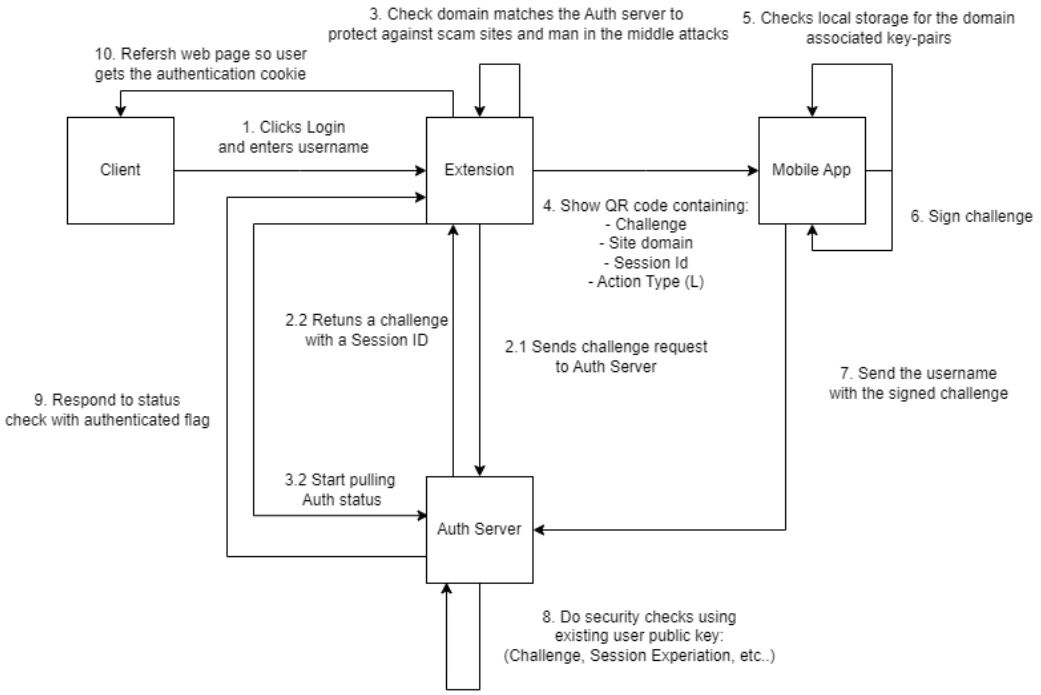

Figure 3.4 : QRAuth login flow.

As can be seen, for the flow to work, no special equipment is needed beyond a device with a camera. Other than that, the flow relies completely on established public key cryptography standards. Before we demonstrate the integration of the above-explained flow into OIDC, it's prudent to go over the individual components needed for each section of the flow to work.

3.2.1 QR code extension

We built a Chrome extension that acts as an example of what an extension that works with our flows needs to have. The extension is just JavaScript so it can be ported to any other browser. The extension was built using Chrome V3 API. The extension has an input field for the user to enter their username, and two buttons for login and register actions. Under them, the QR code is shown to the user. See Figure 3.5 for a look at the extension

Figure 3.5 : Browser extension showing the user a QR code.

The extension uses a JS library called QRCode [29] to help render the QR code. The library was used to reduce boilerplate code, but any other library or custom code be used in its place.

Finally, we would like to point out the URL components of the QR code shown to the user. First, we have "code" which is the challenge sent from the authentication server. Second, the username represents the value the user entered as their user handle. Third, is the session ID as received from the authentication server since it will be forwarded by the mobile app to the authentication server to track the current active session. Lastly, the action type is either an “L” representing login action, or an “R” representing register.

3.2.2 Mobile app

The mobile app was built using React Native to minimize boilerplate code. The mobile app acts as the authenticator and has a very similar responsibility to the FIDO2 authenticator. The first action handled by the mobile app is the generation of the public-private key pair. Second, it handles the signing of the challenge using the user’s private key. Third, the authenticator is responsible for saving the private-public key pairs in secure storage. The way it saves the keys (especially the private key) is left open to each authenticator’s needs[2]. In our implementation, we opted to use the secure storage of the OS running the app. Finally, the app is responsible for communicating

[2]This freedom is one of the main features that us apart from FIDO2, as the user is the one that picks what authenticator they want to use giving them real ownership over how their keys are handled.

with the authentication server to share the user's public key on registration and the signed challenge when requested.

3.3 Integrating the QR Code Flow With OpenID Connect

It's clear where our flow would slot in the OIDC protocol. Instead of using passwords to authenticate users, we can use our proposed protocol. Figure 3.6 shows the registration flow integration. The new steps related to our proposal that are deviations from the "normal" OIDC implementation are in red.

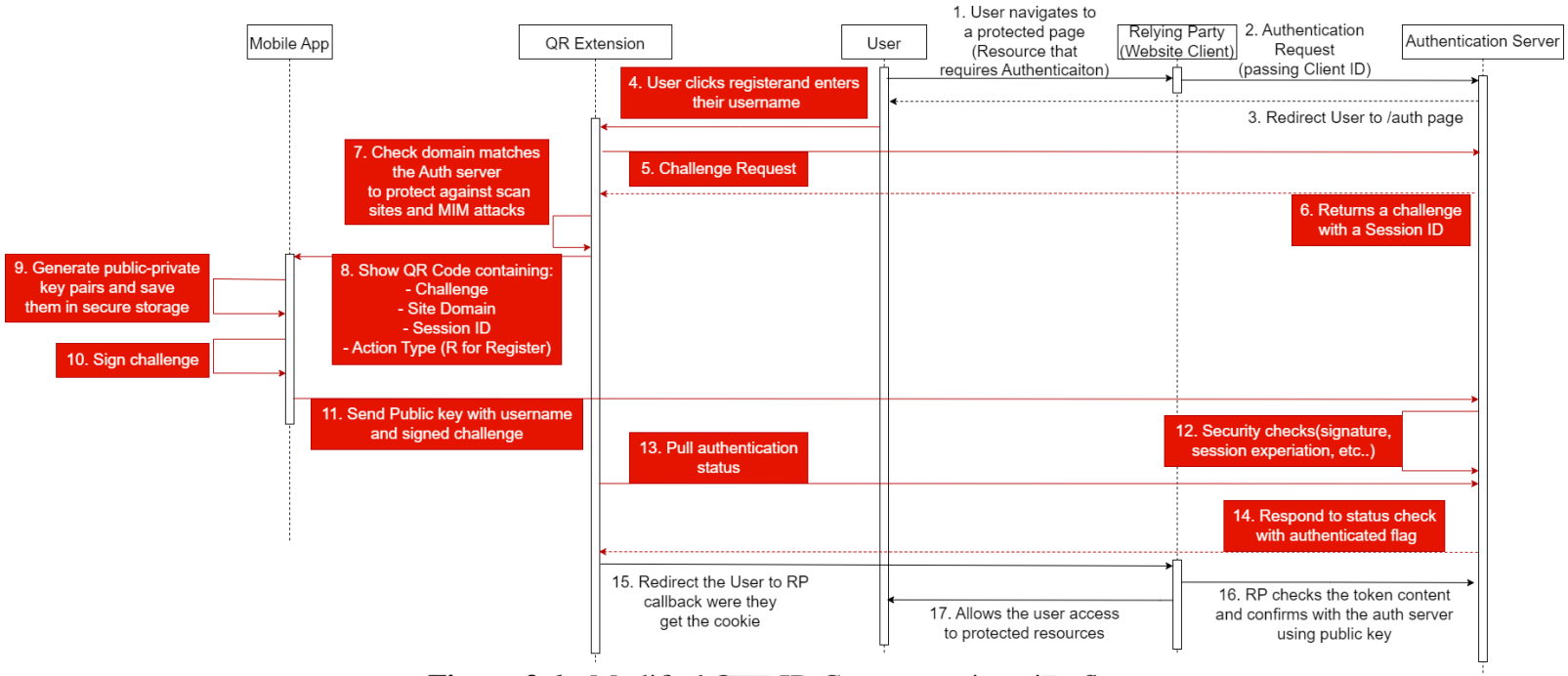

Figure 3.6 : Modified OpenID Connect registration flow.

The flow diverges from the normal OIDC starting from step 4 and returns to the normal flow at step 15. The full steps are as follows:

1. First, as is done in the normal protocol the flow starts when the user navigates to a protected resource.
2. The relying party sends an authentication request to the authentication server containing the relaying party client ID.
3. The authentication server redirects the user to the /auth page.
4. The user is prompted to enter their username in the browser extension.
5. The extension sends a challenge request to the authentication server that contains the username.
6. The authentication server generates a random challenge, associating it with a session ID. Both the session ID and the challenge are returned to the extension.
7. The extension performs security checks to prevent phishing attacks.
8. The extension renders a QR code to the user containing the site domain, the user-chosen username, and the challenge plus session ID received from the authentication server.
9. The extension starts an authentication status-pulling job against the authentication server that runs every half a second.
10. The user then, using the mobile app, reads the QR code.
11. The mobile app parses the QR code, and checks that no key pair exists in the device secure storage for the parsed domain.
12. The mobile app generates the public-private key pairs and saves the private key in the device's secure storage associating it with the site domain.
13. Using the private key, the app signs the received challenge.

14. The signed challenge, with the username, session ID, and public key are then forwarded to the authentication server.

15. The authentication server performs security checks such as session expiration, and signature verification.

16. The authentication server responds to the authentication status pulling request indicating that the user is authenticated.

17. The extension redirects the user to the relaying party callback to get the authentication token.

18. The relying party checks the token content and confirms its authenticity with the authentication server using the public key retrieved from .openid-configuration.

19. The relying party allows the user access to the protected resource.

The login flow is very similar and can be seen in Figure 3.7 and goes as follows:

1. First, as is done in the normal protocol the flow starts when the user navigates to a protected resource.

2. The relying party sends an authentication request to the authentication server containing the relaying party client ID.

3. The authentication server redirects the user to /auth page.

4. The user is prompted to enter their username in the browser extension.

5. The extension sends a challenge request to the authentication server that contains the username.

6. The authentication server generates a random challenge, associating it with a session ID. Both the session ID and the challenge are returned to the extension.

7. The extension performs security checks to prevent phishing attacks.

8. The extension renders a QR code to the user containing the site domain, the user-chosen username, and the challenge plus session ID received from the authentication server.

9. The extension starts an authentication status-pulling job against the authentication server that runs every half a second.

10. The user then, using the mobile app, reads the QR code.

11. The mobile app parses the QR code, and checks that a key pair exists in the device secure storage for the parsed domain.

12. Using the existing private key, the app signs the received challenge.

13. The signed challenge, with the username and session ID is then forwarded to the authentication server.

14. The authentication server performs security checks such as session expiration, and signature verification.

15. The authentication server responds to the authentication status pulling request indicating that the user is authenticated.

16. The extension redirects the user to the relaying party callback to get the authentication token.

17. The relying party checks the token content and confirms its authenticity with the authentication server using the public key retrieved from .openid-configuration.

18. The relying party allows the user access to the protected resource.

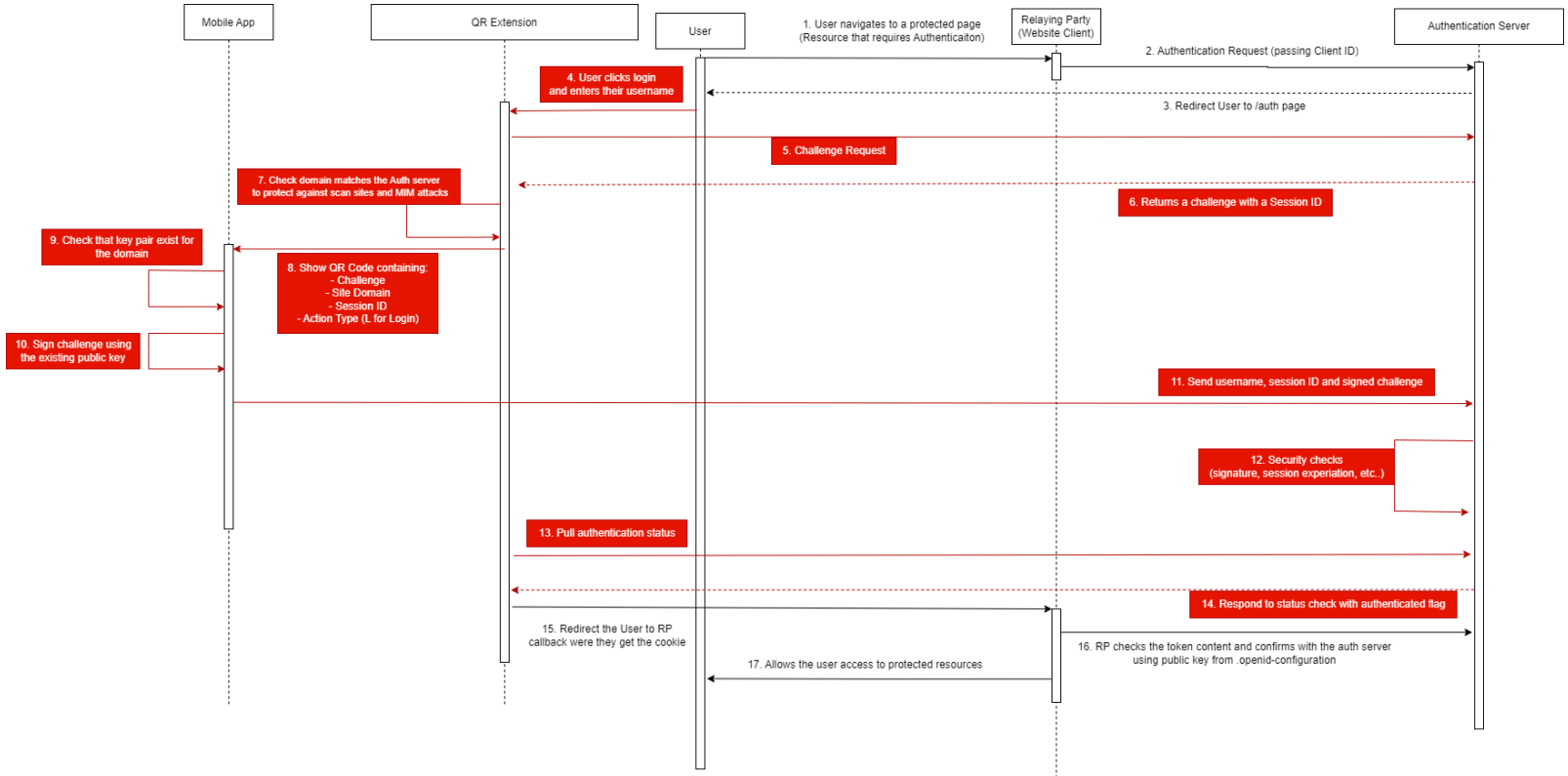

Figure 3.7 : Modified OpenID Connect login flow.

With the flow details out of the way, we can now examine how it works under the hood. First, we will showcase snapshots of the authentication server then we will describe how the relying party works.

3.3.1 Authentication server

The authentication server was built using IdentityServer4 which is an ASP.NET framework that helps with security-related boilerplate. We used the built-in OpenId Connect using tokens template as the baseline for the project and modified it as appropriate. The code generates a random string to act as the challenge and then saves it to the database associating it with the user. The server uses a "SessionExperiationDate" variable to track if the current active session has expired or is still active. The need for a session timer and tracker will be further explored in the security analysis section. The authentication server challenge-response security checks as seen in step 12 in Figure 3.4 and 3.3 include checking that the session has not yet expired by comparing its creation date against the time the response was received as well as verifying the user's signature using RSA algorithm. Finally, the authentication server does normal request validation such as the existence of the user on login, and that the username is not duplicated on registration. The last part of our proposal that the authentication server handles is replying to the authentication status pull request initiated by the browser extension.

3.3.2 Relying party

We built a sample relying party using ASP.NET that only has two basic HTML pages. The first is an unprotected page that anyone can access without having to be authenticated, the second is a "protected resource" page where only authenticated users can access it. This was done to showcase that the authentication is working as expected.

3.4 QRAuth Security Analysis

To reduce the security analysis needed for our proposed flow, we have implemented it as part of the OIDC flow. In essence, what's needed is analyzing the security of the modified parts of the protocol only. First, for both FIDO2 and normal passwords,

the issue of phishing attacks and scam sites is a glaring problem. If a site is able to convince users that it's a genuine site, it can forward the communication from the user to itself to the genuine site conducting an MITM attack. However, in our proposed method, browser extension has the responsibility of checking the genuineness of the visited site before showing the QR code. As a result, our only presumption is not that of the security of the relying party or its resistance to phishing attacks but the security of the user device. If the browser the user is using is secure, the only security issue would be a fake extension imitating a real one - an issue that could be solved with the existence of certified browser stores to authenticate extensions. Second, our proposed protocol supports "credentials" sharing in a way, that both FIDO2 and passwords can not. If a user wants to delegate access to their account to another user (for example the case of allowing a trusted partner access to a personal account) in the case of FIDO2 it's completely not possible. The original device that houses the key pairs associated when the registration happened must be present, and the owner of the account that created the keys using their biometrics must be there to authenticate on the device. Passwords, on the other hand, support permanent account delegation in that the original owner of the account can't revoke access unless the password itself is changed. Our proposed protocol solves this issue by the use of QR codes which are just images that can be easily shared by the person wanting access to the account to the account owner. If the account owner wants to grant access, they can read the QR code using their original device and approve the access. Then, when the session expires, the delegated user can't access the account again without the re-approval of the account owner. Hence, the move to QR code completely solves one of the issues that have plagued practically all previously proposed web authentication protocols. Nonetheless, this issue of account sharing opens the door to replay attacks where the QR code from a valid session is saved and then re-shown to the user on a malicious site. Consequently, we included a session variable (set at 1 minute in our sample app) that can be changed based on the needs of the app owner. Increasing the timer would make account sharing easier but open the door to replay attacks. If the value is set to a very short period (e.g., 5 seconds), replay attacks become practically impossible as the attacker needs to start the authentication process on the genuine site, get the QR code image, send the image

to the account owner to read the QR code, approve the authentication, and for all the other communication explained above to complete before the session expires. A longer session period would account for the delay in getting the QR code from the delegated user to the original account owner. We plan in the future to research more on what the values should be for the session expiration timer for different use cases where the needs for security and usability are different. Third, as the security communication only happens directly between the user's mobile and the authentication server over TLS, our presumptions for security should only extend beyond the user's browser (that is the app being used to navigate the relying party) to the user's mobile device used to read the QR code. The mobile device will handle saving the user's private key, generating signatures, and sending them with the user's public key to the authentication server. Consequently, the security of the flow lies in the user's mobile device. If the device gets hacked, the security of the flow will collapse. Finally, as our proposal is based on public key cryptography, all the security assumptions of digital signatures are applicable. Any risk, as can be explored in the rich literature on digital signatures and public key cryptography is also a risk in our case, as similar to FIDO2.

4. COMPARING PASSWORDS VS FIDO2 VS QRAUTH

The complete comparison between our FIDO2 implementation, our QRAuth proposal, and existing common password-based authentication flows can be seen in Table 4.1 By the term "user familiarity", we mean how familiar a user is with the authentication flow. Passwords are the most familiar type of web authentication method. However, most users are also familiar with QR codes so asking the user to read a QR code to authenticate themselves is not a strange action to most of them. FIDO2 on the other hand requires the user first to select what device to use while WebAuth initializes, and then, only after selecting the correct device from the prompts, to approve a notification on their device that will also require them to re-authenticate themselves to their devices usually with biometrics. Integration with existing websites is the biggest hurdle to FIDO2's widespread adoption. Although in the first half of this project, we tried to minimize the work needed to replace a password in an existing website, still quite some work is needed to add WebAuth to the sites' front-end and update the backend with all the logic needed to handle FIDO2 related security requirements. Our protocol relieves this to a great extent. Although we believe our proposed flow is a better step in the direction of replacing passwords, it still has disadvantages, like FIDO2, in two main areas: account recovery and authentication for mobile applications. The second issue is not an unsolvable one. We leave it as possible future work beyond this project. It is possible to set up a case where if a mobile app needs to authenticate the user, the user can use a pre-shared (during registration) app domain for the authenticator to find the related public key to sign the challenge and complete the authentication. Nonetheless, the issue of account recovery is still a major one, and more research is needed to offer better solutions for public-key-based authentication flows [30]. We also acknowledge that our proposal still requires "special" equipment to work. The first is a camera-enabled device, and the second is the installation of a browser extension. The need for a camera-enabled device is not a serious issue for the vast majority of users with how ubiquitous smart devices have become. However, requiring the users

to download an extension is a hurdle to wide adoption. Further work is needed to determine how much of a limitation this requirement is[1], nonetheless, if our proposal gets adopted by major browser operators then the need for the extension can be lifted as the work being done by the extension can be completely handled by the browser as is done in FIDO2. Lastly, as described in the literature review, any modification to how FIDO works must be accepted by the FIDO alliance, such as the work done by Chegqian [21] where our proposal doesn't have this issue. We don't restrict or even suggest any special requirements for authenticators[2] thus leaving the landscape open for improvements in our proposal by other contributors.

[1] We plan on conducting user studies to measure the usability of our proposal, and part of that is the requirement for users to install a browser extension

[2] Except for the suggestion on session expiration time which can be configured per the requirement of each authentication server operator.

Table 4.1 : Comparing passwords, FIDO2 and QRAuth.

	Password	FIDO2	QRAuth
Security Algorithm	Hash functions	Public key cryptography	Public key cryptography
Account Delegation	Permanent	N/A	Temporary
User Familiarity	Most familiar	Unfamiliar	Familiar
Special Requirements	N/A	FA certified device & software	Any device with camera
Integration With Existing Web Sites	N/A	Hard	Easy
Account Recovery	Possible	N/A	N/A
Susceptible To Phishing Attacks	Yes	No	No
Requires Usages of Specific Protocols	N/A	CTAP & WebAuthn	N/A
Needs Official Support	N/A	FIDO Alliance	N/A
Mobile Apps Authentication	Yes	No	No

5. CONCLUSION

Despite the considerable security enhancements it provides, the FIDO2 authentication protocol has yet to achieve widespread adoption. This is primarily attributable to a multitude of factors, including its limited adaptability and lack of support for account delegation.

In an endeavor to address these limitations, we propose an innovative authentication flow, termed as QRAuth. This novel approach incorporates a browser extension, to help ensure our proposal is phishing attack-resistant.

Our proposal significantly mitigates the adaptability issue inherent in FIDO2. It accomplishes this by necessitating only a device equipped with a QR code reader, thereby enhancing its accessibility and ease of implementation across a broad spectrum of devices.

Moreover, QRAuth introduces the concept of shareable QR code images. This unique feature facilitates temporary account delegation, a functionality that is not supported in FIDO2 at all, and to limited effect in conventional passwords. This implies that users can temporarily delegate access to their accounts, thereby adding an additional layer of security and convenience.

Nonetheless, we acknowledge that the challenge of account recovery persists with our proposed system, akin to FIDO2. In scenarios where users lose access to their accounts, devising a secure and user-friendly recovery method remains an unresolved issue.

As part of our future endeavors, we aim to undertake a formal security analysis of our QRAuth proposal. This will encompass rigorous testing and formal security analysis to ascertain that our system can effectively counter various security threats.

Another promising avenue for our subsequent work involves conducting a user study. This will enable us to compare the usability aspects of QRAuth with other

authentication methods. By collating user feedback and studying user interaction patterns, we can further refine our system and enhance its intuitiveness and user-friendliness.

In conclusion, while FIDO2 offers substantial security advantages, its limitations necessitate the evolution of more adaptable and flexible authentication methods like QRAuth. Through incessant research and development, we aspire to contribute towards making the digital world safer and more secure for all users.

REFERENCES

[1] **Machani, S. and Field, N.** (2022). Choosing FIDO Authenticators for Enterprise Use Cases, *FIDO Alliance White Paper*, FIDO Alliance.

[2] **Biçakci, K. and Uzunay, Y.** (2022). Is FIDO2 passwordless authentication a hype or for real?: A position paper, *15th International Conference on Information Security and Cryptography*, Ankara, Turkey.

[3] **Alliance, F.** *Certified Authenticator Levels*, Retrieved October 23, 2023, from https://fidoalliance.org/certification/authenticator-certification-levels/.

[4] **Tschofenig, H.** (2016). Fixing User Authentication for the Internet of Things (IoT): Integrating FIDO and OAuth into IoT, *Datenschutz und Datensicherheit - DuD*, *40*, 222–224.

[5] **Sharevski, F.** (2022). Phishing with malicious QR codes, *2 European Symposium on Usable Security, 2022*.

[6] **Han, L.**, **Wong, D. and Chao, L.** (2014). Password Cracking and Countermeasures in Computer Security: A Survey.

[7] **Carstens, D.**, **McCauley, P.**, **Malone, L. and Demara, R.** (2004). Evaluation of the human impact of password authentication practices on information security, *Informing Science*, *7*.

[8] **Alkhalil, Z.**, **Hewage, C.**, **Nawaf, L. and Khan, I.** (2021). Phishing Attacks: A Recent Comprehensive Study and a New Anatomy, volume 3, `https://doi.org/10.3389/fcomp.2021.563060`.

[9] **Bonneau, J.**, **Herley, C.**, **Oorschot, P.C.v. and Stajano, F.** (2012). The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes, *2012 IEEE Symposium on Security and Privacy*, pp.553–567.

[10] **Zimmermann, V.**, **Gerber, N.**, **Kleboth, M.**, **Preuschen, A.**, **Schmidt, K. and Mayer, P.** (2018). The Quest to Replace Passwords Revisited – Rating Authentication Schemes.

[11] **Fukumitsu, M.**, **Hasegawa, S.**, **Iwazaki, J.Y.**, **Sakai, M. and Takahashi, D.** (2016). A Proposal of a Password Manager Satisfying Security and Usability by Using the Secret Sharing and a Personal Server, *2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA)*, pp.661–668.

[12] **Ghorbani Lyastani, S.**, **Schilling, M.**, **Neumayr, M.**, **Backes, M. and Bugiel, S.** (2020). Is FIDO2 the Kingslayer of User Authentication? A Comparative Usability Study of FIDO2 Passwordless Authentication, *2020 IEEE Symposium on Security and Privacy (SP)*, pp.268–285.

[13] **Angelogianni, A.**, **Politis, I. and Xenakis, C.** (2021). How many FIDO protocols are needed? Surveying the design, security and market perspectives, *arXiv preprint arXiv:2107.00577*.

[14] **Barbosa, M.**, **Boldyreva, A.**, **Chen, S. and Warnischi, B.** (2022). Provable Security Analysis of FIDO2, *CRYPTO*.

[15] **Guan, J.**, **Li, H.**, **Ye, H. and Zhao, Z.** (2022). *A Formal Analysis of the FIDO2 Protocols*, `https://doi.org/10.1007/978-3-031-17143-7_1`.

[16] **Oh, D.S.**, **Kim, B.H. and Lee, J.K.** (2011). *A Study on Authentication System Using QR Code for Mobile Cloud Computing Environment*, `https://doi.org/10.1007/978-3-642-22333-4_65`.

[17] **Al-Ghaili, A.**, **Kasim, H.**, **Othman, M. and Hashim, W.** (2020). QR code based authentication method for IoT applications using three security layers, *TELKOMNIKA (Telecommunication Computing Electronics and Control)*, *18*, 2004.

[18] **C, P.**, **M, G.**, **Kim and N, A.D.** (2015). Analysis of Secure Authentication System Using QR Code, pp.151–160.

[19] **Kim, Y.G. and Jun, M.S.** (2011). A design of user authentication system using QR code identifying method, *2011 6th International Conference on Computer Sciences and Convergence Information Technology (ICCIT)*, pp.31–35.

[20] **Sung, S.**, **Lee, J.**, **Kim, J.**, **Moon, J. and Won, D.** (2018). Security Analysis of Mobile Authentication Using QR-Codes, volume 5.

[21] **Guo, C.**, **Cai, Q.**, **Wang, Q. and Lin, J.** (2020). Extending Registration and Authentication Processes of FIDO2 External Authenticator with QR Codes, *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pp.518–529.

[22] **Krombholz, K.**, **Frühwirt, P.**, **Rieder, T.**, **Kapsalis, I.**, **Ullrich, J. and Weippl, E.** (2015). QR Code Security – How Secure and Usable Apps Can Protect Users Against Malicious QR Codes, *2015 10th International Conference on Availability, Reliability and Security*, pp.230–237.

[23] **Vidas, T.**, **Owusu, E.**, **Wang, S.**, **Zeng, C.**, **Cranor, L.F. and Christin, N.** (2013). *QRishing: The Susceptibility of Smartphone Users to QR Code Phishing Attacks*, `https://doi.org/10.1007/978-3-642-41320-9_4`.

[24] **Dudheria, R.** (2017). Evaluating Features and Effectiveness of Secure QR Code Scanners, *2017 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, pp.40–49.

[25] **Stefan** (2023). *QR Code Authentication OpenID Connect: A Step Towards Passwordless Security*, `https://shorturl.at/dBCZ1`.

[26] **Fett, D.**, **Kuesters, R. and Schmitz, G.** (2017). *The Web SSO Standard OpenID Connect: In-Depth Formal Security Analysis and Security Guidelines*, `1704.08539`.

[27] **Navas, J. and Beltrán, M.** (2019). Understanding and mitigating OpenID Connect threats, *Computers Security*, *84*.

[28] **Mainka, C.**, **Mladenov, V.**, **Schwenk, J. and Wich, T.** (2017). SoK: Single Sign-On Security — An Evaluation of OpenID Connect, *2017 IEEE European Symposium on Security and Privacy (EuroSP)*, pp.251–266.

[29] **Shim, S.** QRCode.js, Retrieved October 10, 2023, from https://github.com/davidshimjs/qrcodejsqrcodejs.

[30] **Conners, J.**, **Devenport, C.**, **Derbidge, S.**, **Farnsworth, N.**, **Gates, K.**, **Lambert, S.**, **McClain, C.**, **Nichols, P. and Zappala, D.** (2022). Let's Authenticate: Automated Certificates for User Authentication, *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*, The Internet Society, `https://www.ndss-symposium.org/ndss-paper/auto-draft-251/`.

APPENDICES

APPENDIX A : FIDO2 Implementation Code Snippets
APPENDIX B : QRAuth Browser Extension Implementation Code Snippets
APPENDIX C : QRAuth Browser Mobile App Implementation Code Snippets
APPENDIX D : QRAuth Relying Party Implementation Code Snippets
APPENDIX E : QRAuth Authentication Server Implementation Code Snippets

APPENDIX A : FIDO2 Implementation Code Snippets

```
[HttpPost]
[ValidateAntiForgeryToken]
[Route("/pwmakeCredentialOptions")]
0 references
public async Task<JsonResult> MakeCredentialOptions(
    [FromForm] string username,
    [FromForm] string displayName,
    [FromForm] string attType,
    [FromForm] string authType,
    [FromForm] bool requireResidentKey,
    [FromForm] string userVerification)
{
    try
    {
        if (string.IsNullOrEmpty(username))
        {
            username = $"{displayName} (Usernameless user created at {DateTime.UtcNow})";
        }

        var user = new Fido2User
        {
            DisplayName = displayName,
            Name = username,
            Id = Encoding.UTF8.GetBytes(username) // byte representation of userID is required
        };

        // 2. Get user existing keys by username
        var items = await _fido2Storage.GetCredentialsByUsername(username);
        var existingKeys = new List<PublicKeyCredentialDescriptor>();
        foreach (var publicKeyCredentialDescriptor in items)
        {
            existingKeys.Add(publicKeyCredentialDescriptor.Descriptor);
        }

        // 3. Create options
        var authenticatorSelection = new AuthenticatorSelection
        {
            RequireResidentKey = requireResidentKey,
            UserVerification = userVerification.ToEnum<UserVerificationRequirement>()
        };

        if (!string.IsNullOrEmpty(authType))
            authenticatorSelection.AuthenticatorAttachment = authType.ToEnum<AuthenticatorAttachment>();

        var exts = new AuthenticationExtensionsClientInputs() {
            Extensions = true,
            UserVerificationIndex = true,
            Location = true,
            UserVerificationMethod = true,
            BiometricAuthenticatorPerformanceBounds = new
            AuthenticatorBiometricPerfBounds {
                FAR = float.MaxValue,
                FRR = float.MaxValue
            }
        };

        var options = _lib.RequestNewCredential(user, existingKeys, authenticatorSelection, attType.ToEnum<AttestationConveyancePreference>(), exts);

        // 4. Temporarily store options, session/in-memory cache/redis/db
        HttpContext.Session.SetString("fido2.attestationOptions", options.ToJson());

        // 5. return options to client
        return Json(options);
    }
    catch (Exception e)
    {
        return Json(new CredentialCreateOptions { Status = "error", ErrorMessage = FormatException(e) });
    }
}
```

Figure A.1 : Creating authentication policy object.

```
[HttpPost]
[ValidateAntiForgeryToken]
[Route("/pwmakeCredential")]
0 references
public async Task<JsonResult> MakeCredential([FromBody] AuthenticatorAttestationRawResponse attestationResponse)
{
    try
    {
        // 1. get the options we sent the client
        var jsonOptions = HttpContext.Session.GetString("fido2.attestationOptions");
        var options = CredentialCreateOptions.FromJson(jsonOptions);

        // 2. Create callback so that lib can verify credential id is unique to this user
        IsCredentialIdUniqueToUserAsyncDelegate callback = async (IsCredentialIdUniqueToUserParams args) =>
        {
            var users = await _fido2Storage.GetUsersByCredentialIdAsync(args.CredentialId);
            if (users.Count > 0) return false;

            return true;
        };

        // 2. Verify and make the credentials
        var success = await _lib.MakeNewCredentialAsync(attestationResponse, options, callback);

        // 3. Store the credentials in db
        await _fido2Storage.AddCredentialToUser(options.User, new FidoStoredCredential
        {
            Username = options.User.Name,
            Descriptor = new PublicKeyCredentialDescriptor(success.Result.CredentialId),
            PublicKey = success.Result.PublicKey,
            UserHandle = success.Result.User.Id,
            SignatureCounter = success.Result.Counter,
            CredType = success.Result.CredType,
            RegDate = DateTime.Now,
            AaGuid = success.Result.Aaguid
        });

        // 4. return "ok" to the client

        var user = await CreateUser(options.User.Name);
        // await _userManager.GetUserAsync(User);

        if (user == null)
        {
            return Json(new CredentialMakeResult { Status = "error", ErrorMessage = $"Unable to load user with ID '{_userManager.GetUserId(User)}'." });
        }

        //await _userManager.SetTwoFactorEnabledAsync(user, true);
        //var userId = await _userManager.FindByNameAsync(user);

        return Json(success);
    }
    catch (Exception e)
    {
        return Json(new CredentialMakeResult { Status = "error", ErrorMessage = FormatException(e) });
    }
}
```

Figure A.2 : Validating registration response and creating a user in the database.

```
[HttpPost]
[Route("/pwassertionOptions")]
0 references
public async Task<ActionResult> AssertionOptionsPost([FromForm] string username, [FromForm] string userVerification)
{
    try
    {

        var existingCredentials = new List<PublicKeyCredentialDescriptor>();

        if (!string.IsNullOrEmpty(username))
        {
            var identityUser = await _userManager.FindByNameAsync(username);
            var user = new Fido2User
            {
                DisplayName = identityUser.UserName,
                Name = identityUser.UserName,
                Id = Encoding.UTF8.GetBytes(identityUser.UserName) // byte representation of userID is required
            };

            if (user == null) throw new ArgumentException("Username was not registered");

            // 2. Get registered credentials from database
            var items = await _fido2Storage.GetCredentialsByUsername(identityUser.UserName);
            existingCredentials = items.Select(c => c.Descriptor).ToList();
        }

        var exts = new AuthenticationExtensionsClientInputs() {
            SimpleTransactionAuthorization = "FIDO",
            GenericTransactionAuthorization = new TxAuthGenericArg {
                ContentType = "text/plain",
                Content = new byte[] { 0x46, 0x49, 0x44, 0x4F }
            },
            UserVerificationIndex = true,
            Location = true,
            UserVerificationMethod = true
        };

        // 3. Create options
        var uv = string.IsNullOrEmpty(userVerification) ? UserVerificationRequirement.Discouraged : userVerification.ToEnum<UserVerificationRequirement>();
        var options = _lib.GetAssertionOptions(
            existingCredentials,
            uv,
            exts
        );

        // 4. Temporarily store options, session/in-memory cache/redis/db
        HttpContext.Session.SetString("fido2.assertionOptions", options.ToJson());

        // 5. Return options to client
        return Json(options);
    }

    catch (Exception e)
    {
        return Json(new AssertionOptions { Status = "error", ErrorMessage = FormatException(e) });
    }
}
```

Figure A.3 : Creating authentication policy object.

```
[HttpPost]
[Route("/pwmakeAssertion")]
0 references
public async Task<JsonResult> MakeAssertion([FromBody] AuthenticatorAssertionRawResponseWithReturnUrl clientResponse)
{
    try
    {
        // check if we are in the context of an authorization request
        var context = await _interaction.GetAuthorizationContextAsync(clientResponse.ReturnUrl);
        // 1. Get the assertion options we sent the client
        var jsonOptions = HttpContext.Session.GetString("fido2.assertionOptions");
        var options = AssertionOptions.FromJson(jsonOptions);

        // 2. Get registered credential from database
        var creds = await _fido2Storage.GetCredentialById(clientResponse.Id);

        if (creds == null)
        {
            throw new Exception("Unknown credentials");
        }

        // 3. Get credential counter from database
        var storedCounter = creds.SignatureCounter;

        // 4. Create callback to check if userhandle owns the credentialId
        IsUserHandleOwnerOfCredentialIdAsync callback = async (args) =>
        {
            var storedCreds = await _fido2Storage.GetCredentialsByUserHandleAsync(args.UserHandle);
            return storedCreds.Exists(c => c.Descriptor.Id.SequenceEqual(args.CredentialId));
        };

        // 5. Make the assertion
        var res = await _lib.MakeAssertionAsync(clientResponse, options, creds.PublicKey, storedCounter, callback);

        // 6. Store the updated counter
        await _fido2Storage.UpdateCounter(res.CredentialId, res.Counter);

        var identityUser = await _userManager.FindByNameAsync(creds.Username);
        if (identityUser == null)
        {
            throw new InvalidOperationException($"Unable to load user.");
        }
        await _events.RaiseAsync(new UserLoginSuccessEvent(creds.Username, identityUser.Id, identityUser.UserName, clientId: context?.Client.ClientId));

        // only set explicit expiration here if user chooses "remember me".
        // otherwise we rely upon expiration configured in cookie middleware.
        AuthenticationProperties props = null;

        // issue authentication cookie with subject ID and username
        var isuser = new IdentityServerUser(identityUser.Id)
        {
            DisplayName = identityUser.UserName
        };

        await HttpContext.SignInAsync(isuser);
        await _signInManager.SignInAsync(identityUser, isPersistent: false);

        // 7. return OK to client
        return Json(res);
    }
    catch (Exception e)
    {
        return Json(new AssertionVerificationResult { Status = "error", ErrorMessage = FormatException(e) });
    }
}
```

Figure A.4 : Verifying user's signature and authentication.

```javascript
async function handleRegisterSubmit(event) {
    event.preventDefault();

    let username = this.username.value;
    let displayName = this.displayName.value;

    // possible values: none, direct, indirect
    let attestation_type = "none";
    // possible values: <empty>, platform, cross-platform
    let authenticator_attachment = "";

    // possible values: preferred, required, discouraged
    let user_verification = "required";

    // possible values: true,false
    let require_resident_key = false;

    // prepare form post data
    var data = new FormData();
    data.append('username', username);
    data.append('displayName', displayName);
    data.append('attType', attestation_type);
    data.append('authType', authenticator_attachment);
    data.append('userVerification', user_verification);
    data.append('requireResidentKey', require_resident_key);

    // send to server for registering
    let makeCredentialOptions;
    try {
        makeCredentialOptions = await fetchMakeCredentialOptions(data);

    } catch (e) {
        console.error(e);
        let msg = "Something wen't really wrong";
        showErrorAlert(msg);
    }

    console.log("Credential Options Object", makeCredentialOptions);

    if (makeCredentialOptions.status !== "ok") {
        console.log("Error creating credential options");
        console.log(makeCredentialOptions.errorMessage);
        showErrorAlert(makeCredentialOptions.errorMessage);
        return;
    }

    // Turn the challenge back into the accepted format of padded base64
    makeCredentialOptions.challenge = coerceToArrayBuffer(makeCredentialOptions.challenge);
    // Turn ID into a UInt8Array Buffer for some reason
    makeCredentialOptions.user.id = coerceToArrayBuffer(makeCredentialOptions.user.id);

    makeCredentialOptions.excludeCredentials = makeCredentialOptions.excludeCredentials.map((c) => {
        c.id = coerceToArrayBuffer(c.id);
        return c;
    });

    if (makeCredentialOptions.authenticatorSelection.authenticatorAttachment === null) makeCredentialOptions.authenticatorSelection.authenticatorAttachment = undefined;

    console.log("Credential Options Formatted", makeCredentialOptions);

    Swal.fire({
        title: 'Registering...',
        text: 'Tap your security key to finish registration.',
        imageUrl: "/images/securityKey.min.svg",
        showCancelButton: true,
        showConfirmButton: false,
        focusConfirm: false,
        focusCancel: false
    });
```

Figure A.5 : Registration code written in JavaScript.

```
document.getElementById('signin').addEventListener('submit', handleSignInSubmit);

async function handleSignInSubmit(event) {
    event.preventDefault();

    let username = this.username.value;

    // possible values: preferred, required, discouraged
    let user_verification = "required";

    // prepare form post data
    var formData = new FormData();
    formData.append('username', username);
    formData.append('userVerification', user_verification);

    // send to server for registering
    let makeAssertionOptions;
    try {
        var res = await fetch('/pwassertionOptions', {
            method: 'POST', // or 'PUT'
            body: formData, // data can be `string` or {object}!
            headers: {
                'Accept': 'application/json'
            }
        });

        makeAssertionOptions = await res.json();
    } catch (e) {
        showErrorAlert("Request to server failed", e);
    }

    console.log("Assertion Options Object", makeAssertionOptions);

    // show options error to user
    if (makeAssertionOptions.status !== "ok") {
        console.log("Error creating assertion options");
        console.log(makeAssertionOptions.errorMessage);
        showErrorAlert(makeAssertionOptions.errorMessage);
        return;
    }

    // todo: switch this to coercebase64
    const challenge = makeAssertionOptions.challenge.replace(/-/g, "+").replace(/_/g, "/");
    makeAssertionOptions.challenge = Uint8Array.from(atob(challenge), c => c.charCodeAt(0));

    // fix escaping. Change this to coerce
    makeAssertionOptions.allowCredentials.forEach(function (listItem) {
        var fixedId = listItem.id.replace(/\_/g, "/").replace(/\-/g, "+");
        listItem.id = Uint8Array.from(atob(fixedId), c => c.charCodeAt(0));
    });

    console.log("Assertion options", makeAssertionOptions);

    Swal.fire({
        title: 'Logging In...',
        text: 'Tap your security key to login.',
        imageUrl: "/images/securitykey.min.svg",
        showCancelButton: true,
        showConfirmButton: false,
        focusConfirm: false,
        focusCancel: false
    });

    // ask browser for credentials (browser will ask connected authenticators)
    let credential;
    try {
        credential = await navigator.credentials.get({ publicKey: makeAssertionOptions })
    } catch (err) {
        showErrorAlert(err.message ? err.message : err);
    }
```

Figure A.6 : Login code written in JavaScript.

APPENDIX B : QRAuth Browser Extension Implementation Code Snippets

```
        fetch( input: `${baseUrl}/generate-qr-challange/${username}`, requestOptions) Promise<Response>
            .then(response : Response => response.json()) Promise<any>
            .then(data => {
                const code : number | string = data.code;
                const sessionId : string = data.sessionId;
                const url : string = `${useLocalMode ? authServerUrl : tab.url}?c=${code}&u=${username}&s=${sessionId}&a=${actionType}`;

                new QRCode(document.getElementById( elementId: "qrcode"), {
                    text: url,
                    width: 128,
                    height: 128,
                    colorDark: "#000000",
                    colorLight: "#ffffff"
                });

                // Start polling authentication status
                pollInterval = pollAuthenticationStatus(username, sessionId, baseUrl);
            })
            .catch(error => {
                console.warn(error);
            });
    });
}
});
```

Figure B.1 : Getting the challenge from the authentication server.

```
document.addEventListener( type: "DOMContentLoaded", listener: function () : void {
    const useLocalMode : boolean = true;
    // Default Auth Server URL. Used as an example for the demo. For production this value is dynamicllly set based on the active site
    const authServerUrl : string = "https://localhost:5001/";
    const apiEndPoint : string = 'api/qRCodeAuthentication';
    const loginButton : HTMLElement = document.getElementById( elementId: "login-button");
    const registerButton : HTMLElement = document.getElementById( elementId: "register-button");
    let pollInterval;
    let redirectUrl;

    loginButton.addEventListener( type: "click", listener: function (e : MouseEvent ) : void {
        e.preventDefault();
        generateQRCode( actionType: "l");
    });

    registerButton.addEventListener( type: "click", listener: function (e : MouseEvent ) : void {
        e.preventDefault();
        generateQRCode( actionType: "r");
    });

    1 usage  ± Ahmad Nawar Droubi
    function pollAuthenticationStatus(username, sessionId, baseUrl, activeTabId) : number {
        return setInterval( handler: () : void => {
            fetch( input: `${baseUrl}/get-session-status?username=${username}&sessionId=${sessionId}`) Promise<Response>
                .then(response : Response => response.json()) Promise<any>
                .then(data => {
                    if (data.authenticated === true) {
                        console.log('User is authenticated!');
                        clearInterval(pollInterval);
                        chrome.tabs.update({url: redirectUrl.replace("/signin-oidc", "")})
                    }
                })
                .catch(error => {
                    console.error('Error occurred during authentication status polling:', error);
                });
        }, timeout: 500); // Polling interval: 0.5 second
    }

    2 usages  ± Ahmad Nawar Droubi
    function generateQRCode(actionType) : void {
        chrome.tabs.query({ active: true, currentWindow: true }, function (tabs) : void {
            const tab = tabs[0];
            const tabUrl : URL = new URL(tab.url);
            const redirectUrlParam : URL = new URL(tabUrl.searchParams.get("ReturnUrl"), tabUrl.origin);
            redirectUrl = redirectUrlParam.searchParams.get("redirect_uri");
            const usernameInput : HTMLElement = document.getElementById( elementId: "username");
            const username = usernameInput.value;
            /* if(useLocalMode && tab.url !== authServerUrl){
                const errorMessageDiv = document.getElementById("error-message");
                errorMessageDiv.innerText = "The current webpage appears to be a none genuine site. Please contact your admin";
                return;
            } */
            const baseUrl : string = (useLocalMode ? authServerUrl : tab.url) + apiEndPoint;

            let requestOptions : {body: null, credentials: string, headers: {…}, method: string, mode: string, referrerPolicy: string}
                "headers": {
                    "accept": "*/*",
                },
                "referrerPolicy": "no-referrer",
```

Figure B.2 : Generating the QR-code.

APPENDIX C : QRAuth Mobile App Implementation Code Snippets

```
export const handleAuthAction = async (
  url,
  username,
  challenge,
  sessionId,
  actionType,
) : Promise<void> => {
  if (actionType === ACTION_TYPES.register) {
    await handleRegister(url, username, challenge, sessionId);
  } else {
    await handleLogin(url, username, challenge, sessionId);
  }
};
1 usage  Ahmad Nawar Droubi
export const handleRegister = async (url, username, challenge, sessionId) : Promise<void> => {
  const keys : string  = await generateKeys(url);
  console.log(`Public key: ${keys}`);
  const signedChallenge : string  = await signChallenge(url, challenge);
  console.log(signedChallenge);
  // Result validated here https://8gwifi.org/rsasignverifyfunctions.jsp with SHA512WithRSA
  await submitPublicKey(username, keys, signedChallenge, sessionId);
};

1 usage  Ahmad Nawar Droubi
export const handleLogin = async (url, username, challenge, sessionId) : Promise<void> => {
  const signedChallenge : string  = await signChallenge(url, challenge);
  console.log(signedChallenge);

  await submitChallenge(username, signedChallenge, sessionId);
};
```

Figure C.1 : The logic that executes upon reading the QR Code.

APPENDIX D : QRAuth Relying Party Implementation Code Snippets

```
using Microsoft.AspNetCore.Authentication.Cookies;
using Microsoft.AspNetCore.Authentication.OpenIdConnect;
using System.IdentityModel.Tokens.Jwt;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllersWithViews();

builder.Services.AddAuthentication(options =>
{
    options.DefaultScheme = "Cookies";
    options.DefaultAuthenticateScheme = "Cookies";
    options.DefaultChallengeScheme = "oidc";
}).AddCookie()
.AddOpenIdConnect("oidc", options =>
{
    options.Authority = "https://localhost:5001";
    options.RequireHttpsMetadata = false;
    options.SignInScheme = "Cookies";
    options.ClientId = "mvc";
    options.SaveTokens = true;
});

var app = builder.Build();

JwtSecurityTokenHandler.DefaultInboundClaimTypeMap.Clear();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    // The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();
app.UseAuthentication();
app.UseAuthorization();
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();
```

Figure D.1 : Relying party authentication middle-ware.

APPENDIX E : QRAuth Authentication Server Implementation Code Snippets

```
[HttpGet("generate-qr-challange/{username}")]
0 references
public async Task<IActionResult> GenerateQRChallange(string username)
{
    byte[] randomBytes = new byte[128];
    using (var rng = new RNGCryptoServiceProvider())
    {
        rng.GetBytes(randomBytes);
    }

    string base64String = Convert.ToBase64String(randomBytes);
    string randomSessionID = Guid.NewGuid().ToString().Substring(0, 10);

    Random random = new Random();
    int length = random.Next(43, 128);
    var challange = base64String.Substring(0, length);

    var existingUser = await _context.QRCodeUsers.FirstOrDefaultAsync(x => x.UserName == username);

    if (existingUser == null)
    {
        var newUser = new QRCodeUser
        {
            UserName = username,
            LastChallenge = challange,
            SessionId = randomSessionID,
            SessionEexpirationDate = DateTime.UtcNow.AddHours(SessionTimeOutHours),
            IsAuthenticated = false
        };

        await _context.QRCodeUsers.AddAsync(newUser);
    } else
    {
        existingUser.LastChallenge = challange;
        existingUser.SessionId = randomSessionID;
        existingUser.SessionEexpirationDate = DateTime.UtcNow.AddHours(SessionTimeOutHours);
        existingUser.IsAuthenticated = false;
        _context.QRCodeUsers.Update(existingUser);
    }

    await _context.SaveChangesAsync();

    return new OkObjectResult(new { code = challange, sessionId = randomSessionID });
}
```

Figure E.1 : Generating the challenge.

```
[HttpGet("generate-qr-challange/{username}")]
0 references
public async Task<IActionResult> GenerateQRChallange(string username)
{
    byte[] randomBytes = new byte[128];
    using (var rng = new RNGCryptoServiceProvider())
    {
        rng.GetBytes(randomBytes);
    }

    string base64String = Convert.ToBase64String(randomBytes);
    string randomSessionID = Guid.NewGuid().ToString().Substring(0, 10);

    Random random = new Random();
    int length = random.Next(43, 128);
    var challange = base64String.Substring(0, length);

    var existingUser = await _context.QRCodeUsers.FirstOrDefaultAsync(x => x.UserName == username);

    if (existingUser == null)
    {
        var newUser = new QRCodeUser
        {
            UserName = username,
            LastChallenge = challange,
            SessionId = randomSessionID,
            SessionEexpirationDate = DateTime.UtcNow.AddHours(SessionTimeOutHours),
            IsAuthenticated = false
        };

        await _context.QRCodeUsers.AddAsync(newUser);
    } else
    {
        existingUser.LastChallenge = challange;
        existingUser.SessionId = randomSessionID;
        existingUser.SessionEexpirationDate = DateTime.UtcNow.AddHours(SessionTimeOutHours);
        existingUser.IsAuthenticated = false;
        _context.QRCodeUsers.Update(existingUser);
    }

    await _context.SaveChangesAsync();

    return new OkObjectResult(new { code = challange, sessionId = randomSessionID });
}
```

Figure E.2 : Handling the user registration request.

```
[Route("verify-challenge")]
[HttpPost]
0 references
public async Task<IActionResult> VerifyChallenge(VerifyChallengeModel model)
{
    var existingUser = await _context.QRCodeUsers.FirstOrDefaultAsync(x => x.UserName == model.Username);

    if (existingUser == null || existingUser.SessionId != model.SessionId)
    {
        return NotFound();
    }

    if (existingUser.SessionEexpirationDate < DateTime.UtcNow)
    {
        return BadRequest(new
        {
            error = "Session Expired"
        });
    }

    if (string.IsNullOrEmpty(existingUser.PublicKey))
    {
        return BadRequest("User hasn't submitted a public key yet.");
    }


    var isValidSignature = VerifySignature(existingUser.LastChallenge, model.SignedChallenge, existingUser.PublicKey);

    if (!isValidSignature)
    {
        return BadRequest("Signature Not Valid");
    }

    existingUser.IsAuthenticated = true;
    _context.QRCodeUsers.Update(existingUser);
    await _context.SaveChangesAsync();

    return Ok();

}
```

Figure E.3 : Logging in the user.

```
2 references
public static bool VerifySignature(string challenge, string signature, string publicKeyPem)
{
    // Decode the PEM-encoded public key
    var publicKeyBytes = Convert.FromBase64String(publicKeyPem
        .Replace("-----BEGIN RSA PUBLIC KEY-----", "")
        .Replace("-----END RSA PUBLIC KEY-----", "")
        .Replace("\n", ""));

    // Create an RSA object and initialize it with the public key
    using var rsa = RSA.Create();
    rsa.ImportRSAPublicKey(publicKeyBytes, out _);

    // Decode the signature from base64
    var signatureBytes = Convert.FromBase64String(signature);

    // Compute the SHA512 hash of the original message
    var sha512 = new SHA512Managed();
    var hashBytes = sha512.ComputeHash(Encoding.UTF8.GetBytes(challenge));

    // Verify the signature using RSA
    var verified = rsa.VerifyHash(hashBytes, signatureBytes, HashAlgorithmName.SHA512, RSASignaturePadding.Pkcs1);

    return verified;
}
```

Figure E.4 : Verifying the user's signature.

```
[Route("get-session-status")]
[HttpGet]
0 references
public async Task<IActionResult> GeChallange(string username, string sessionId)
{
    var user = await _context.QRCodeUsers.FirstOrDefaultAsync(x => x.UserName == username);

    if (user  == null || user.SessionId != sessionId)
    {
        return NotFound();
    }

    if (user.SessionEexpirationDate < DateTime.UtcNow)
    {
        return BadRequest(new {
            error = "Session Expired"
        });
    }

    // issue authentication cookie with subject ID and username
    var isuser = new IdentityServerUser(user.Id)
    {
        DisplayName = user.UserName
    };

    var responseMessage = "Session pending approvel";
    var authenticated = false;

    if (user.IsAuthenticated)
    {
        await HttpContext.SignInAsync(isuser);
        await _signInManager.SignInAsync(user, isPersistent: false);
        responseMessage = "User authenticated";
        authenticated = true;
    }
    return Ok(new { authenticated = authenticated,  message = responseMessage});
}
```

Figure E.5 : The code that handles authentication status requests.

CURRICULUM VITAE

Name SURNAME: Ahmet DROBI

EDUCATION:

- **B.Sc.:** 2020, T.C. İstanbul Kültür Üniversitesi, Engineering, Computer Science Engineering

PROFESSIONAL EXPERIENCE AND REWARDS:

- 2018-2019 Full Stack Developer at AR Root.
- 2020-2021 Full Stack Engineer at Adam IcT.
- 2021-2022 Analyst Software Developer at NMQ Digital
- 2022-2023 Software Developer at Protect Group.
- 2023 Expert Software Developer at NMQ Digital.

PUBLICATIONS, PRESENTATIONS AND PATENTS ON THE THESIS:

- **Drobi A.**, Bicakci, K. (2023). QRAuth: A Secure and Accessible Web Authentication Alternative to FIDO2, *16th International Conference on Security and Cryptology,* Ankara, Turkey.