mAuth: Secure Authorization and Authentication Protocol for Native Apps

Master’s thesis in Computer Systems and Networks

FREDRIK HAMREFORS
ADAM TÖRNKVIST

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's thesis 2024

mAuth: Secure Authorization and Authentication Protocol for Native Apps

FREDRIK HAMREFORS
ADAM TÖRNKVIST

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

mAuth: Secure Authorization and Authentication Protocol for Native Apps
FREDRIK HAMREFORS
ADAM TÖRNKVIST

© Fredrik Hamrefors, Adam Törnkvist, 2024.

Supervisor: Miranda Aldrin, Omegapoint
Supervisor: Gerardo Schneider, Dept. of Computer Science and Engineering
Examiner: Romaric Duvignau, Dept. of Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2024

mAuth: Secure Authorization and Authentication Protocol for Native Apps
Fredrik Hamrefors
Adam Törnkvist
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

OAuth and OIDC are well-established for handling user authentication and authorization, which are industry standards today. However, the user experience of native mobile apps remains a challenge due to the use of browser redirection. It is hard for users to determine if they can trust the web page that pops up during the login process. To improve on this, we have designed a protocol called mAuth that performs user authentication and authorization on mobile phones without the use of browser redirection. This protocol follows the best current practice (BCP) of OAuth and the FAPI standard. The analysis of the protocol showed that mAuth follows the BCPs for OAuth and FAPI through the use of attestation, demonstrating proof of possession (DPoP) with the client instance key and following the basis of the authorization code flow. From the analysis of the user experience of the theoretical protocol, which is based on our point of view, we found that it achieves the goal of better user experience. It also provides flexibility for the developer as they can choose between three different flows depending on their security and user experience demands.

Keywords: OAuth, OIDC, Authentication, Authorization, Security, Native, App, login, mAuth.

Acknowledgements

We want to express our gratitude to Miranda Aldrin, our technical supervisor at Omegapoint, who was always ready to answer questions and provide help. We would also like to thank Tobias Ahnoff and Kasper Karlsson for their feedback and suggestions regarding the protocol. A big thanks to Adrian Bjugård for helping set up the network for intercepting traffic from the apps, to our supervisor at Chalmers, Gerardo Schneider, for help with the structure of the report, and to our examiner, Romaric Duvignau for guiding the project as a whole. Also, thanks to our opponents, Albin Pansell and Simon Riis, for the good feedback that helped to make the thesis better understood by our audience.

Fredrik Hamrefors and Adam Törnkvist, Gothenburg, May 2024

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

AAT	API Access Token
API	Application Programming Interface
AS	Authorization Server
BCP	Best Current Practice
CAT	Client Attestation Token
CIBA	Client-Initiated Backchannel Authentication
CSRF	Cross-site Script Request Forgery
DoS	Denial of Service
DPoP	Demonstrate Proof of Possession
DPoPP	Demonstrate Proof of Possession Proof
FAPI	Financial-Grade API
FIDO2	Fast Identity Online 2
HAAPI	Hypermedia Authentication API
HTTPS	Hypertext Transfer Protocol Secure
JSON	Java Script Object Notation
JWKS	JSON Web Key Sets
JWS	JSON Web Signature
JWT	JSON Web Token
mAuth	Mobile Authentication
MFA	Multi-factor Authentication
mTLS	Mutual Transport Layer Security
OIDC	OpenID Connect
OTP	One-time Password
PKCE	Proof Key for Code Exchange
RFC	Request For Comments
RO	Resource Owner
ROPC	Resource Owner Password Credentials
RS	Resource Server
RSA	Rivest-Shamir-Adleman
TLS	Transport Layer Security
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UX	User Experience

Contents

List of Acronyms ix

List of Figures xv

List of Tables xvii

1 Introduction 1
- 1.1 Brief History of OAuth . . . 1
- 1.2 Problem Statement . . . 3
- 1.3 Approach . . . 4
- 1.4 Limitations . . . 5
- 1.5 Outline . . . 5

2 Background 7
- 2.1 OAuth 2.1 . . . 7
  - 2.1.1 Access Tokens . . . 8
  - 2.1.2 Claims . . . 8
  - 2.1.3 Authorization Code Flow with PKCE . . . 8
  - 2.1.4 PKCE . . . 10
  - 2.1.5 Resource Owner Password Credentials Flow . . . 10
  - 2.1.6 Demonstrating Proof of Possession . . . 11
  - 2.1.7 OIDC . . . 13
  - 2.1.8 CIBA . . . 13
- 2.2 FIDO2 . . . 15
- 2.3 Attestation . . . 17
  - 2.3.1 Secure Storage on Mobile Phone . . . 17
  - 2.3.2 Android Attestation . . . 17
  - 2.3.3 iOS Attestation . . . 19
- 2.4 Multi-Factor Authentication . . . 21
- 2.5 OAuth 2.0 Best Current Security Practices . . . 21
  - 2.5.1 Protecting Redirect-Based Flows . . . 21
  - 2.5.2 Token Replay Prevention . . . 22
  - 2.5.3 Access Token Privilege Protection . . . 22
  - 2.5.4 Client Authentication . . . 23
  - 2.5.5 Other Recommendations . . . 23
- 2.6 FAPI 2.0 Security Profile . . . 23
  - 2.6.1 Network Layer Protections . . . 23

2.6.2 Requirements of The Actors . . . 24
2.6.3 Cryptography and Secrets . . . 24
2.6.4 Security Considerations . . . 24
2.6.5 FAPI CIBA . . . 25
2.7 OAuth 2.0 Threat Model . . . 25
2.7.1 Assumptions About Attacker . . . 25
2.7.2 Attacker Obtains The Refresh Token . . . 25
2.7.3 Attacker Obtains Access Token . . . 26
2.7.4 Attacker Obtains Authorization Code . . . 26

3 Related Work **27**
3.1 HAAPI . . . 27
3.2 OAuth 2.0 Attestation-Based Client Authentication . . . 30
3.3 OAuth 2.0 for First-Party Applications . . . 31
3.4 User Experience . . . 32

4 Our Approach **33**
4.1 Pre-Study . . . 33
4.2 App Analysis . . . 33
4.3 Protocol Design . . . 34

5 App Analysis **35**
5.1 App A . . . 35
5.2 App B . . . 35
5.3 App C . . . 35
5.4 App D . . . 35
5.5 App E . . . 36
5.6 Analysis . . . 36

6 mAuth - Our Proposed Protocol **37**
6.1 Detailed Description . . . 37
6.2 Integrated ROPC . . . 40
6.3 Integrated CIBA . . . 41
6.4 Integrated FIDO2 . . . 42
6.5 Client Verification for User . . . 43
6.6 Security Analysis . . . 44
6.7 mAuth in Pseudocode . . . 44
6.8 Comparison with Regular OAuth Code Flow . . . 45

7 Discussion **47**
7.1 Protocol Analysis . . . 47
7.2 Security Analysis . . . 47
7.2.1 ROPC . . . 48
7.2.2 PKCE . . . 48
7.2.3 DPoP . . . 48
7.2.4 Attestation . . . 49
7.3 Pseudocode Analysis . . . 49

7.3.1 Complexity of Pseudocode 49
7.3.2 Library Security 49
7.4 Client Verification for User 50
7.5 User Experience 50
7.6 Downsides and Constraints 51

8 Conclusion **53**
8.1 Future Work 54

Bibliography **55**

A Security Analysis **I**

B Pseudocode **XI**
B.1 Client Frontend XI
B.2 Client Backend XII
B.3 Authorization Server XIII
B.4 API XV

List of Figures

1.1 The world without OAuth 2
1.2 The world with OAuth 3

2.1 OAuth 2.1 Authentication Code flow with PKCE. The figure is based on the one found at [8] 9
2.2 The flow of DPoP 11
2.3 The CIBA flow in poll mode 14
2.4 The flow of FIDO2 16
2.5 The flow of Attestation in Android 18
2.6 The flow of Attestation in iOS 20

3.1 The flow of HAAPI 28
3.2 OAuth 2.0 Attestation Based Client Authentication 31
3.3 OAuth 2.0 for First-Party Applications 32

6.1 The flow of mAuth 38
6.2 The flow of ROPC in mAuth 40
6.3 The flow of CIBA in mAuth 41
6.4 The flow of FIDO2 in mAuth 43
6.5 The flow for the user to verify the client 44

List of Tables

6.1 Protocol comparison between our three protocol versions and the regular OAuth Code flow. 45

1
Introduction

OAuth 2.1, together with Open ID Connect (OIDC), is well-established for handling user authentication and authorization. It has become an industry standard, including for financial grade applications. However, the user experience (UX) for native mobile apps remains a challenge. The standard flow of OAuth when a user tries to log into a mobile application works by having the user be redirected via the web browser in order to perform the login. The problem is that it is difficult for a user to know if they can trust the web page that pops up, leading to a bad user experience. Thus, many organisations would prefer keeping the user in the app instead of having them be redirected through a web browser and still have the security benefits of the OAuth and OIDC Code flow.

With this in mind, developers also need to follow the best current security practice (BCP) when developing their applications. This could be realised by designing a protocol that does not compromise the BCP when using OAuth 2.1 together with OIDC and still keep the user in the app. This would enhance the user experience and simplify the login process for mobile apps.

On the technical side, one can see that OAuth and OIDC are designed with the ability to perform redirections in order to perform authorization or authentication [48]. The OAuth Request For Comments (RFC) provides examples of how to perform redirections but also clearly states that these are not requirements and that "any other method available via the user-agent to accomplish this redirection is allowed and is considered to be an implementation detail" [29]. The OIDC core specification also says that redirection is implementation-dependent. This opens the possibility of designing a new protocol where there is no redirection performed via the browser and instead keeps the user in the app.

1.1 Brief History of OAuth

Mobile apps have been present since the launch of the first smartphone in 1994, but it was only in the launch of the first iPhone in 2007 that the concept of mobile apps became widespread [27]. Just the next year, Apple launched its app store, which contained over 500 apps. The number of apps would increase dramatically in the years to come. In 2007, OAuth 1.0 was released. OAuth 1.0 was not designed for mobile apps, which made it difficult to utilise this authorization standard [28]. It was only when OAuth 2.0 was released in 2012 that mobile apps could also use the

industry standard for authorization.

OAuth was originally created to allow third-party apps access to certain resources without showing the app your credentials. For example, some photo editing app might want to access your Google photos. You want the app to be able to see the photos but do not want to give the app your Google password. This is where OAuth came in. With OAuth, you could log in directly to Google and allow the photo app to access your Google photos. Previously, you would give the photo app your Google password, and then the app would use your password to log in to your account and take the photos from there, as can be seen in Figure 1.1.

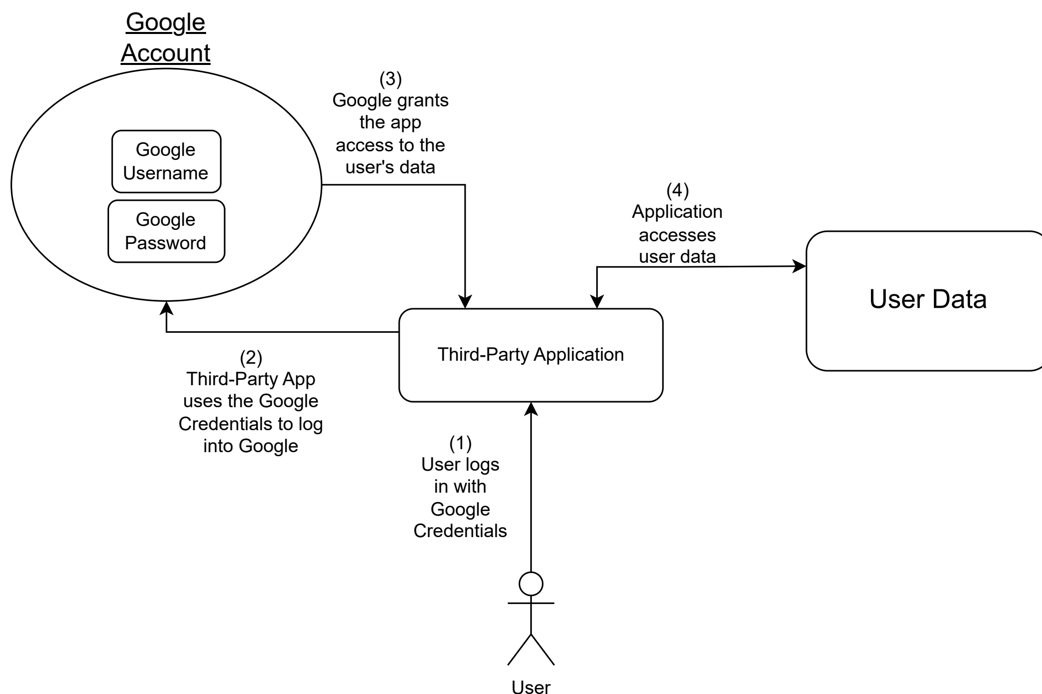

Figure 1.1: The world without OAuth

The app might even save your password to make it easier to log in next time. The user would have to trust that the app did not abuse the password or that the app would not be hacked and your password leaked that way. There was also no way to limit access to your data. In the previous example, you would want the app to see your photos but not your email. Before OAuth, the app either had full access or no access. OAuth solved all of this by introducing access delegation, allowing a third party to access protected resources on a user's behalf.

In Figure 1.2, one can see that with OAuth, the user logs in via the web browser. After a successful login, the Google server grants the application access to the requested files. During this process, the third-party application could not see or in any way read the user's username and password. This means that if the app is malicious or hacked, your password will not be leaked.

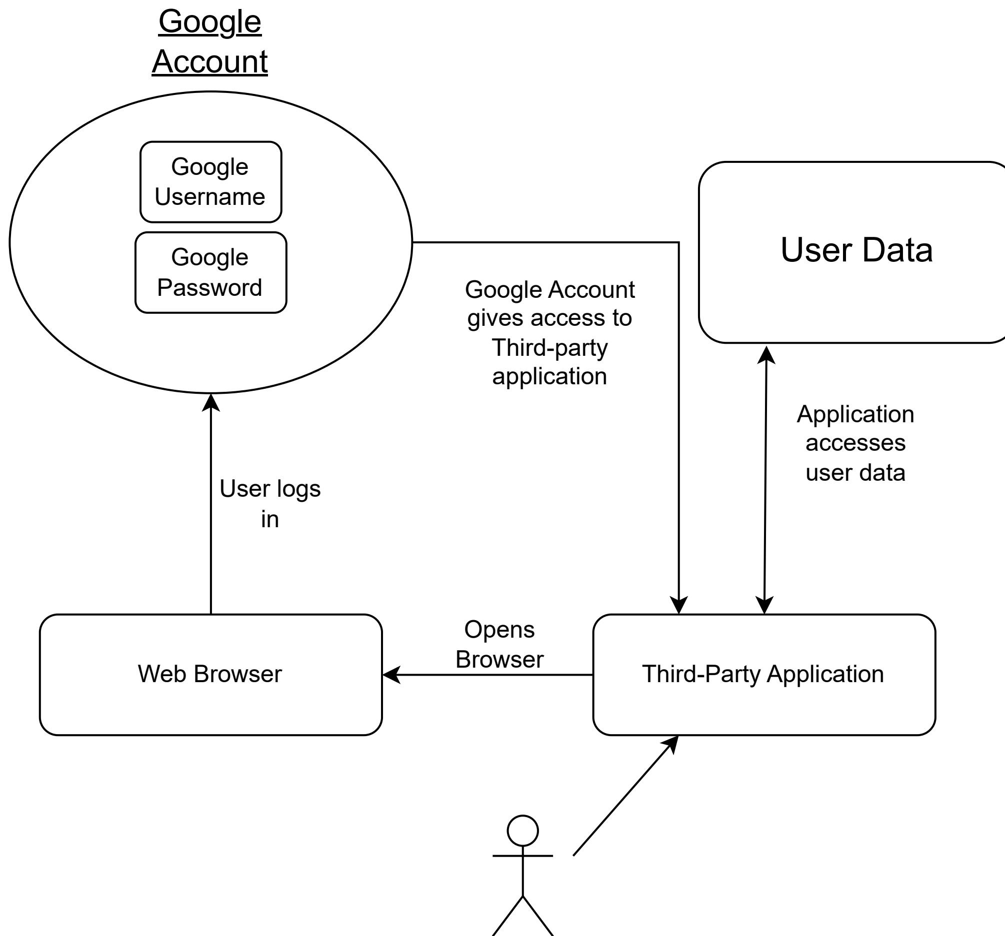

Figure 1.2: The world with OAuth

Since its inception, OAuth has evolved beyond its original purpose. There are now different flows for different situations. The protocol itself is also evolving as work is being done on OAuth 2.1. Currently, this specification is only in the draft stages, but one can observe that flows present in OAuth 2.0 e.g. the implicit flow will be removed in the 2.1 version as it is deemed to be insecure. This shows how the security landscape evolves.

1.2 Problem Statement

This thesis reviews the current OAuth 2.1 with OIDC protocol and its use of browser redirection. The security benefits and the BCPs are taken into account when designing a new protocol that does not utilise browser redirection. This gives the user a more native login experience and, thus, a better UX while still keeping the login process secure. This protocol was also security analysed according to these BCP.

Thus, the goal of this thesis is to design a protocol that reaps the security benefits of OAuth and OIDC without using browser redirection. We want to answer if it is possible to design such a protocol that follows the BCPs for OAuth that utilises decoupled flows like the Client Initiated Backchannel Authentication (CIBA) flow and supports strong user authentication such as BankID and Fast Identity Online 2 (FIDO2). We also want to perform a detailed security analysis of the protocol with the BCPs of OAuth in mind and investigate how the user experience is affected compared to the regular OAuth code flow.

The thesis can be summarised in the following research questions:

1. Is it possible to design an authentication/authorization protocol that follows the BCPs without browser redirection?

2. Can such a protocol also implement decoupled flows such as CIBA and FIDO2?

3. How is user experience affected compared to regular OAuth?

1.3 Approach

This thesis is organised into three phases. The first is about acquiring relevant knowledge, studying solutions to a similar problem, and observing how actual in-use apps perform authentication/authorization. The second consists of designing a theoretical protocol that performs authentication/authorization without browser redirection. Lastly, we conduct a security analysis of the protocol with regard to the BCPs, analyse the user experience, and also write pseudocode in order to illustrate the logical flow of the protocol in code.

In the first phase, we study the current research field of OAuth and the relevant RFCs, as well as proposed solutions to similar problems. We use this knowledge during the study of real apps in order to analyse the security aspect of their authentication/authorization process.

The design phase consists of first identifying the parts of the OAuth protocol that become vulnerable for native apps when browser redirection is not used. Then, we apply the knowledge gained from the previous phase to mitigate these security threats.

The security analysis is based on the BCPs studied during the first phase. For each practice, we ask ourselves the following questions: "Do we follow this practice?" and "Is this practice within the scope of this thesis?". If we do not follow the practice and it is within the scope, we analyse if a modification to the protocol is needed or not in order to still be secure. The analysis of the user experience is from our point of view and not from a real user's view of testing and an app. The reason for this is presented in Section § 1.4.

1.4 Limitations

In this thesis, we have limited ourselves to only designing a theoretical protocol and not a fully implemented practical version. The protocol has instead been written in pseudocode, which represents how the protocol functions. When possible, we have included existing libraries and where to use them.

We have also limited the thesis to only focus on mobile phones and not other devices such as computers or IoT devices.

To evaluate the user experience of the protocol, we have not done a test on real users since the protocol currently is theoretical and not practical. Instead, we have included a discussion regarding the user experience compared to regular OAuth 2.0 based on our interpretation of how the UX is affected.

1.5 Outline

In Chapter 2, we talk about the concepts surrounding OAuth that are used in our final results, such as Demonstration Proof of Possession (DPoP), attestation, CIBA, and FIDO2. Chapter 3 talks about the Hypermedia Authentication API made by Curity, the techniques used by them, how they solve the same problem, and other current research. In Chapter 4, we describe the methodology of our research and the design process of our protocol. Chapter 5 goes over and studies other apps that handle the login process natively. Chapter 6 presents our protocol, how the general flow works, and how the three different specific integrated login processes, CIBA, Resource Owner Password Credentials flow (ROPC), and FIDO2, work with our protocol. It also contains a security analysis of the protocol and a description of the pseudocode. Chapter 7 contains discussions about our findings in Chapter 6. Finally, Chapter 8 contains our conclusion. Appendix A contains the full security analysis, and Appendix B contains the written pseudocode.

2

Background

This chapter describes the background theory needed to understand the problem and how our proposed protocol work. This includes a detailed description of the OAuth 2.1 protocol and the relevant flows, as well as the BCP regarding OAuth, the FAPI 2.0 Security Profile, and the OAuth 2.0 Threat Model.

2.1 OAuth 2.1

OAuth 2.1 is an authorization protocol that allows applications and websites to access data hosted by other web apps with the user's consent. The advantage of this is that the user's credentials are never revealed to any third-party application or website [11] [23]. While OAuth does provide authorization, it does not provide authentication. Authentication is the process where a user verifies they are who they claim to be. Authorization is when a particular user can access restricted resources or services, given that they have been authenticated. In order to perform authentication, OAuth needs to be used together with OpenID Connect (OIDC).

The authorization framework surrounding OAuth consists of an authorization layer where the protocol separates the participating actors into four roles. The roles are the following:

Resource Owner (RO): The actor that grants access to the restricted resource, usually the end user.

Resource Server (RS): The server that hosts the restricted resource. Usually, some API.

Client: The application or website requesting access to the restricted resource on behalf of the Resource Owner.

Authorization Server (AS): The server that authenticates the Resource owner and issues an access token after the user has been authorized that can be used to access the restricted resource.

When using the OAuth protocol, a client would first initiate a request to access some restricted resource belonging to the resource owner and hosted by the resource server. This could be an application on a mobile phone that wants to read a contact list or access some photos. This request from the client is sent to the authorization server, which then asks the resource owner to authenticate themselves with their credentials. If the authentication succeeds and the resource owner is authorized to

access these resources, the authorization server provides the client with an access token that it can use to access the requested resources by sending that token to the resource server that is hosting that resource.

The OAuth authorization framework consists of two different types of endpoints: the authorization endpoint and the token endpoint. An endpoint is a pre-defined point where the communication for some action is set to occur. The authorization endpoint obtains authorization from a resource owner to access a restricted resource. The token endpoint is used by an application to get a token.

2.1.1 Access Tokens

Clients use the access tokens in OAuth 2.1 to get authorized to access some protected resource [11]. These tokens contain different types of permissions or claims, known as scopes, which define what type of actions or rights the entity holding this token has when accessing the protected resource. These tokens have different formats, and one commonly used is the JSON web token (JWT) format. A JWT token consists of three different fields [10]:

JOSE Header: Stands for JSON Object Signing and Encryption. This field contains the metadata about the token type and what cryptographic algorithms are used to secure its content, i.e., what hashing algorithm is used.

JWS Payload: This field contains statements about the user and additional attributes known as claims, such as what type of permissions the user has. The different claims are described in Section § 2.1.2.

JWS Signature: This field is used to validate the token, which consists of verifying that the sender of the token is who they claim to be and that the token has not been modified along the way. To create this signature, the header, the payload and a secret are signed using a cryptographic algorithm specified in the header, e.g., HMAC SHA256.

2.1.2 Claims

The claims contained in the payload of a JWT token are information regarding an entity [9]. It could be a claim about the identity of a user or if the user is an administrator or not. There exist two types of claims: registered and custom. Registered claims are claims defined by the JWT specification to ensure interoperability with external or third-party applications. Custom claims are claims that are not registered and defined by other developers.

2.1.3 Authorization Code Flow with PKCE

The Authorization Code flow with Proof Key for Code Exchange (PKCE) is one of the standard flows of OAuth 2.1. It is the primary flow to use for native applications, and the flow works in 11 steps, as can be seen in Figure 2.1 [37]:

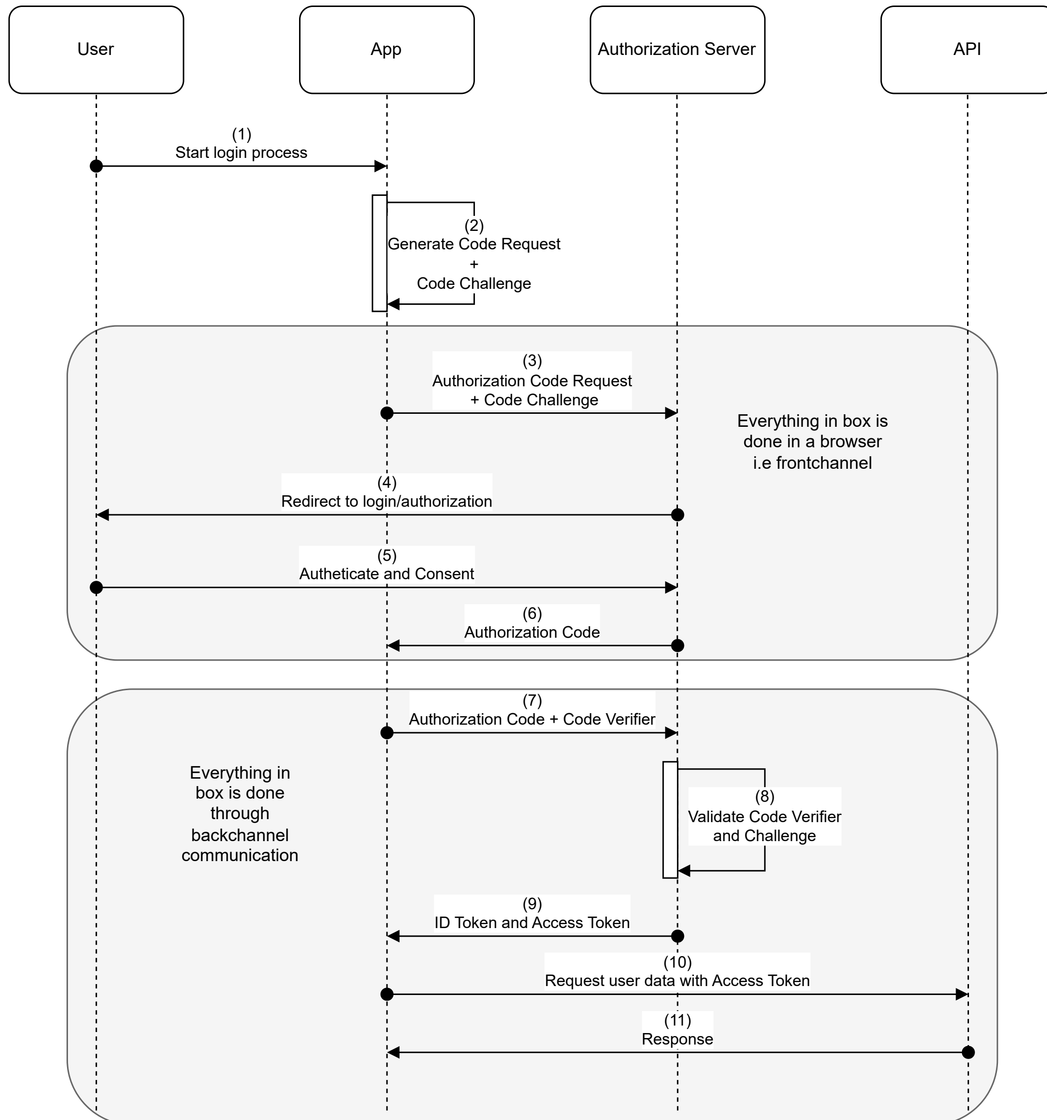

Figure 2.1: OAuth 2.1 Authentication Code flow with PKCE. The figure is based on the one found at [8]

(1) The user clicks the login link in the app.

(2) The app generates a Code verifier and a Code Challenge for PKCE.

(3) A web browser window is opened in the app, and an authorization code request, the code challenge, a callback URI, and a scope of what data the app wants to access are sent to the authorization server using HTTPS.

(4) The user is then redirected to a login/authorization prompt in the browser.

(5) The user authenticates and consents to share the data specified in the scope.

(6) The authorization server sends an authorization code back to the app using HTTPS.

(7) The app requests an access token and an ID token by sending the authorization code and the Code verifier generated in step (2) to the authorization server.

(8) The authorization server validates the code verifier with the code challenge sent in step (3).

(9) The authorization server sends back an ID token and an access token to the app using HTTPS. The ID token can then be used to authenticate the user in the app.

(10) The app requests the resource server with the access token to obtain the data specified in the scope.

(11) The resource server checks the access token and responds to the app with the corresponding data specified in the scope.

The communication between the app, the AS and the API can be performed in two ways: frontchannel and backchannel communication. Frontchannel communication means the communication is happening via a browser over the web. This is easy to use and allows a degree of separation between the client and the user's login information. Backchannel, on the other hand, is when the communication is done directly between the client and server and is therefore more secure and less vulnerable to interception.

2.1.4 PKCE

Proof Key for Code Exchange, or PKCE, extends the Authorization Code flow presented in OAuth 2.1 [42]. PKCE aims to prevent Cross-Site Request Forgery (CSRF) attacks and code injection attacks. What PKCE does is that for every authorization request, the client generates a secret, also known as a code verifier. When the client makes the authorization code request, it sends a hashed version of this secret, known as a code challenge. When exchanging the authorization code for an access token later, the client sends the original non-hashed secret. By doing this, even if the authorization code is intercepted, one would need the secret to obtain the requested token.

2.1.5 Resource Owner Password Credentials Flow

The Resource Owner Password Credentials (ROPC) Flow is one of the standard flows defined in OAuth 2.0 [29]. It is often used for legacy and or migration reasons. This flow works by first having the resource owner, usually the user, provide the

client with their username and password. Then, the client makes a token request by sending a post request to the authorization server containing these credentials. In this request, the client also authenticates with the authorization server. The server then authenticates the client, validates the credentials, and issues an access token if valid.

This flow has now been deprecated and is scheduled to be removed in the OAuth 2.1 Authorization Framework [30]. Also, the OAuth 2.0 best current practice says that this flow MUST NOT be used as it insecurely exposes the resource owner's credentials to the client [32]. This increases the attack vector as credentials can leak in more places than the authorization server. Also, this flow is not designed to work with more robust authentication mechanisms such as multi-factor authentication.

2.1.6 Demonstrating Proof of Possession

Demonstrating Proof of Possession (DPoP) is a mechanism that an authorization server could use to constrain tokens to a specific client [25]. The client has a public/private key pair, and the tokens are then bound to the public key of the client's key pair and the client can prove that it is in possession of the private key by including a DPoP header in an HTTP request. By having the client prove the possession of the private key, the authorization server can get some assurance that this client is the one for whom the tokens were issued. This mechanism is used in our protocol to sender constrain the tokens, which means that these tokens can only be used by this particular sender.

The flow of DPoP can be seen in Figure 2.2 and the process is described in detail below.

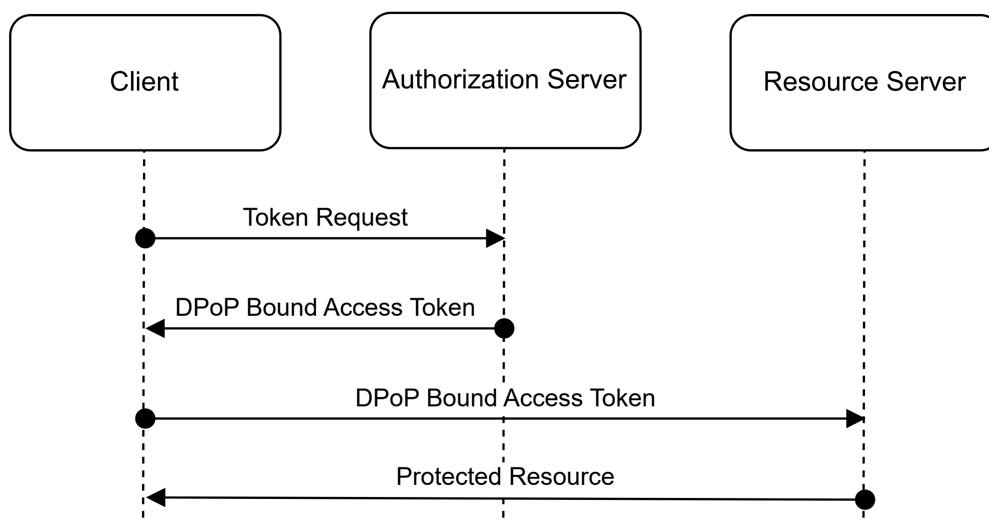

Figure 2.2: The flow of DPoP

During the DPoP flow, when a client requests an authorization code or a token from the authorization server, a DPoP proof is attached to the HTTP header. This proof is a JWT created by the client, showing that it possesses the private key used to sign the JWT. The authorization server then cryptographically binds the code/token to the corresponding public key that the client claims in the DPoP proof.

The DPoP proof consists of a header and a payload. These two fields have some required parameters, which are described below. The header has the following parameters:

`typ`: The type of token. For DPoP, it should always be set to `dpop+jwt`.

`alg`: The JWS asymmetric digital signature algorithm used.

`jwk`: The public key. This parameter can include several subparameters, including key type and key id.

The DPoP proof payload consists of the following parameters:

`jti`: A unique identifier for that DPoP proof JWT.

`htm`: The HTTP method used.

`htu`: The HTTP target URI of the request to which the JWT is attached.

`iat`: The creation timestamp of the JWT.

`ath`: The hash of the access token for which this proof is used.

`nonce`: The nonce provided by the DPoP-Nonce header.

To verify a DPoP proof, the receiving server needs to perform the following checks in any order it would like:

1. There is only one DPoP request in the header field.
2. The aforementioned header field is a single and well-formed JWT.
3. All the parameters of the header field and payload described above are contained in the JWT.
4. The `typ` parameter is set to `dpop+jwt`.
5. The `alg` is not "none" but an algorithm registered and supported by the local policy.
6. The signature of the JWT verifies with the public key contained in the `jwk` parameter.
7. The key contained in the `jwk` parameter is not a private key.
8. The method in the `htm` parameter is the same as the current HTTP request.
9. The `htu` parameter matches the HTTP URI value in the current HTTP request.
10. The value in the `nonce` parameter matches the nonce provided by the server.
11. The time in the `iat` parameter is within an acceptable time frame.
12. The value in the `ath` parameter matches the token sent together with the proof.
13. The public key for which the token is bound to match the public key in the proof.

Then, to use the code/token, the client must again prove that it possesses the private key by including the DPoP proof in the HTTP header of the request containing

the code/token. The public key must also be included in the request sent to the server. The server verifies the proof as described above. If any of these checks fail, the server refuses the request.

For each HTTP request, the client must provide a new, unique DPoP proof. It is also worth noting that a DPoP proof cannot act alone as an access control or authentication method. However, it is a valuable tool to use in conjunction with other mechanisms.

2.1.7 OIDC

OpenID Connect (OIDC) is an identity authentication protocol that is used to make users able to verify who they are when logging in to an application or website [12]. OIDC is an extension of OAuth that provides authentication in the form of an ID token that can be used to authenticate the user. This ID token is in the same format as the access tokens, the JWT format. It is used together with OAuth, but OAuth only provides authorization, while OIDC provides the possibility for authentication. The OIDC specification defines a set of standard claims similar to OAuth. However, it is also possible to define custom claims if the standard claims do not cover the necessary information. In our protocol, OIDC is used to provide authentication.

2.1.8 CIBA

Client-Initiated Backchannel Authentication, or CIBA, is an OIDC authentication flow that can authenticate a user without having end-user interaction from the consumption device [21]. The flow used in CIBA is also known as a decoupled flow. The communication between the client and the authentication server is done without redirection through the user's browser. When using CIBA, the authentication can be initiated by one device and carried out on another. By doing this, one client could obtain tokens from an authorization server for a given user on another device after the same user has been authenticated using another device. A user could also use the same device that the client is currently running on to perform the authentication.

In CIBA, there are three different flows, but they all accomplish the same thing. They are poll mode, ping mode, and push mode. The recommended flow to use is poll mode [49], illustrated in Figure 2.3 and described step by step below.

(1) The client makes an authentication request to the Backchannel Authentication Endpoint at the authorization server.

(2) The authorization server responds with an ACK message that includes a unique identifier for that request and a parameter indicating the maximum lifetime of the transaction.

(3) The authorization server performs the user authentication by using a mobile device to interact with the user. The authorization server chooses the authen-

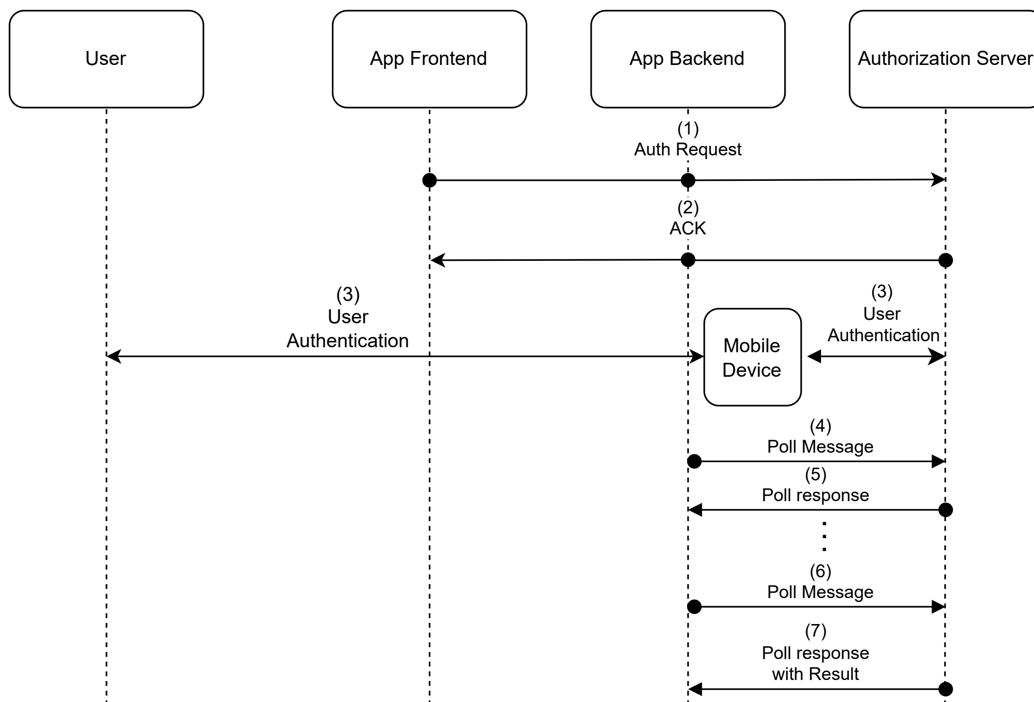

Figure 2.3: The CIBA flow in poll mode

tication method used depending on the situation.

(4) The backend starts regularly polling the authorization server to check if the authentication process is done.

(5) The authorization server tells the backend that the process is not done.

(6) The backend keeps polling until it gets a result from the authentication process.

(7) The authorization server sends the result of the authentication process to the backend.

Steps 1 through 3 are the same for all modes, but the actual token delivery differs depending on the mode used. In ping mode, the authorization server sends a ping message to the client, indicating that the client can retrieve the authentication outcome. The client sends a token request to the authorization server, and the server responds with a token if the authentication is successful. In push mode, the client waits until the authentication process is done and then the authorization server sends a token response to the client backend.

2.2 FIDO2

FIDO2 is an authentication standard created by the FIDO alliance and enables the public client to utilise strong authentication using public key cryptography [51] [45]. FIDO2 enables an application to perform multi-factor and passwordless authentication using, e.g., fingerprint scanning and facial recognition.

In FIDO2, there are three different actors [14]. Those are the following:

Relying Party: The entity that has the application that uses FIDO2. In native apps, the relying party is the authorization server that authenticates the users.

FIDO2 Client: This is where FIDO2 is implemented, usually in the form of software. In the case of native apps, then the client is the FIDO2 client.

Authenticator: This entity creates and stores a user's credentials for a specific relying party. The authenticator could exist in hardware or software and be integrated into the same device as the FIDO2 client, user agent, or on a separate device.

FIDO2 consists of two ceremonies (OAuth uses flow instead, but they mean the same thing): registration and authentication. The two ceremonies look similar and can be depicted in Figure 2.4. The registration ceremony is used to register the credentials for a user for a particular relying party at the authenticator. The authentication ceremony uses the authenticator to authenticate a user at a specific relying party. Below follows a description of each step in the ceremonies:

(1)(4) The user initiates the ceremony, and the FIDO2 client retrieves the login prompt from the relying party and displays it to the user.

(5) The user clicks the register or login buttons. The register button leads to the registration ceremony, and the login button leads to the authentication ceremony.

(6) The FIDO2 client requests a challenge from the relying party that will be used later to verify the response from the authenticator.

(7) The relying party responds with a challenge to the FIDO2 client. This challenge is a cryptographic nonce. For the registration ceremony, the relying party also sends a "create" request to indicate that it wants to create new credentials for a user. For the authentication ceremony, the relying party also sends a "get" request to indicate that it wants to authenticate a user.

(8) For both ceremonies, the FIDO2 client proxies and sends the request and the challenge to the authenticator. In the request, the client also includes the origin information of the request.

(9) For the registration flow, the authenticator prompts the user to create new cre-

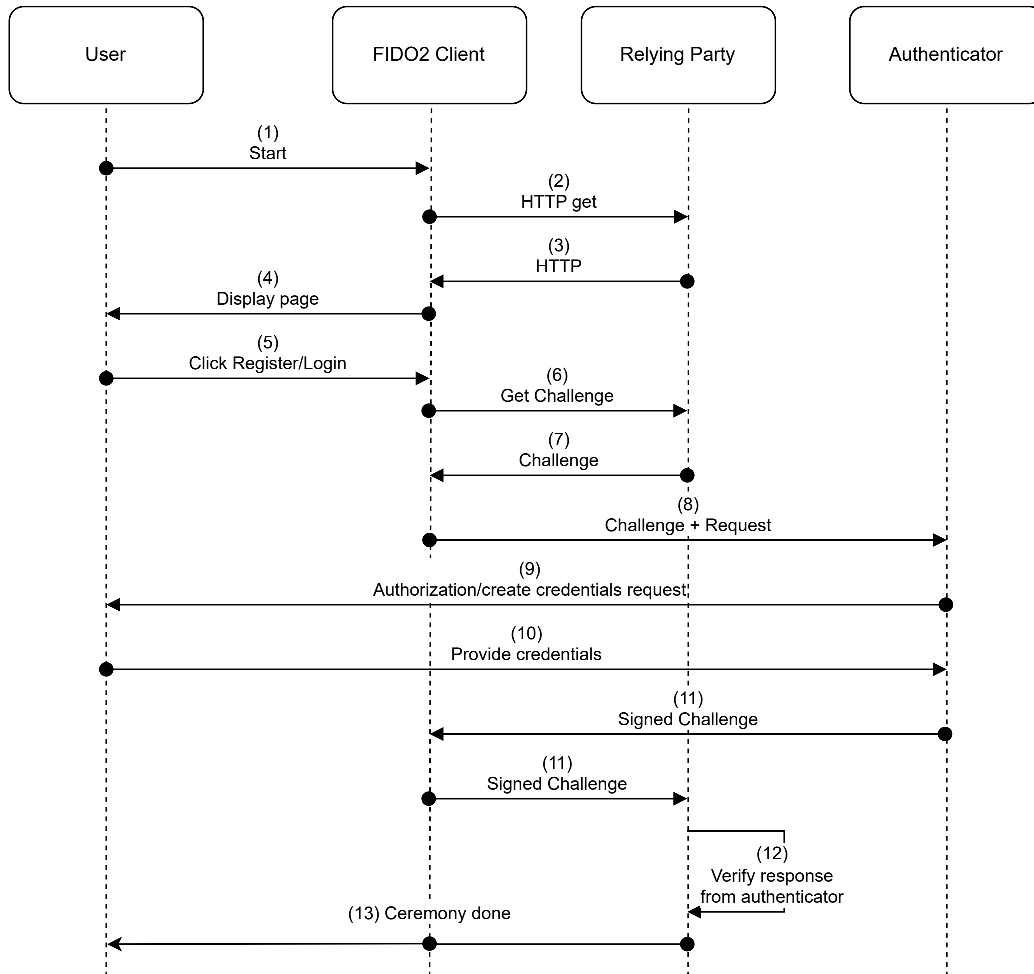

Figure 2.4: The flow of FIDO2

dentials at the requesting relying party. For the authentication ceremony, the authenticator asks if the user wants to authenticate to the requesting relying party.

(10) The user provides their credentials to the authenticator. This could be, e.g., a PIN or biometric credentials. In the case of registration, the authenticator creates a new set of credentials based on this in the form of a public/private key pair.

(11) The authenticator creates a response for the relying party, including the origin info, the challenge, and, in the case of registration, the public key created in step 10. This key is not present in the authentication ceremony response. In both cases, the authenticator signs the response using the private key and sends the response to the relying party, proxied via the FIDO2 client.

(12) The relying party verifies the response from the authenticator. This includes validating the authenticity of the authenticator, the integrity of the response, and the fact that the origin is the same as expected. Validating the origin prevents phishing attacks. The relying party stores the public key in the response for the registration ceremony. This public key is then used in the authentication ceremony to authenticate the user.

(13) The relying party indicates to the FIDO2 client and, in turn, the user the result of either the registration or authentication ceremonies.

2.3 Attestation

Attestation in the context of mobile apps and devices is the process of verifying the provenance of an app and checking that the app is running on a genuine device. This is achieved by using certificates and digital signatures and verifying those using a public key infrastructure. This is an important part of mobile security and has led to both Google and Apple creating their own systems for providing attestation. The system Google uses is the Play Integrity API [1], and Apple uses the Device Check and App Attest protocol [5]. Our protocol uses this to verify the app to the authorization server.

2.3.1 Secure Storage on Mobile Phone

During the attestation process for both operating systems, cryptographic keys are utilised and stored securely on the mobile phone. Android and iOS have their own mechanism for storing some data elements securely. On Android, it is called the KeyStore system; on iOS, it is called the Secure Enclave [18] [3]. When these systems are used during a process in an application, the application would not have direct access to the key. However, it could instead use the system to access the key and perform cryptographic operations. This makes it harder for an attacker to steal the keys.

2.3.2 Android Attestation

Android mobile phones have a public and private key stored in a secure element in the device, either in hardware or software, that is not exposed to any application. This private key can be used to sign challenges an authorization server provides. It can also obtain a certificate for an app from the KeyStore system bound to the mobile phone's private key. This means that the certificate can be used to verify the signature of some data signed by the private key. The certificate contains information about the package name of the app as well as the hash of the certificate used to sign the app when it was published to Google Play. Through the combination of these properties, the app's provenance can be verified. The reliability of the attestation is dependent on the trustworthiness of the operating system.

To ensure that the operating system of the device is reliable, the previously mentioned certificate also contains information about the execution environment, whether the private key used for signing is stored in hardware or software, and information about whether the essential parts of the operating system are verified. If all of these properties of the certificate hold, then the sandbox that the app is running in is secure, which means that the app is not rooted or a clone. This means that one can trust the relationship between the certificate regarding the app's package name and the signing key. This, in turn, means that the provenance of the app is guaranteed as long as the certificate is valid. To use the Play Integrity API, one needs a phone manufactured after August 2017 to have the hardware required [16].

Figure 2.5 and the description below explain in detail how attestation in Android is performed step by step.

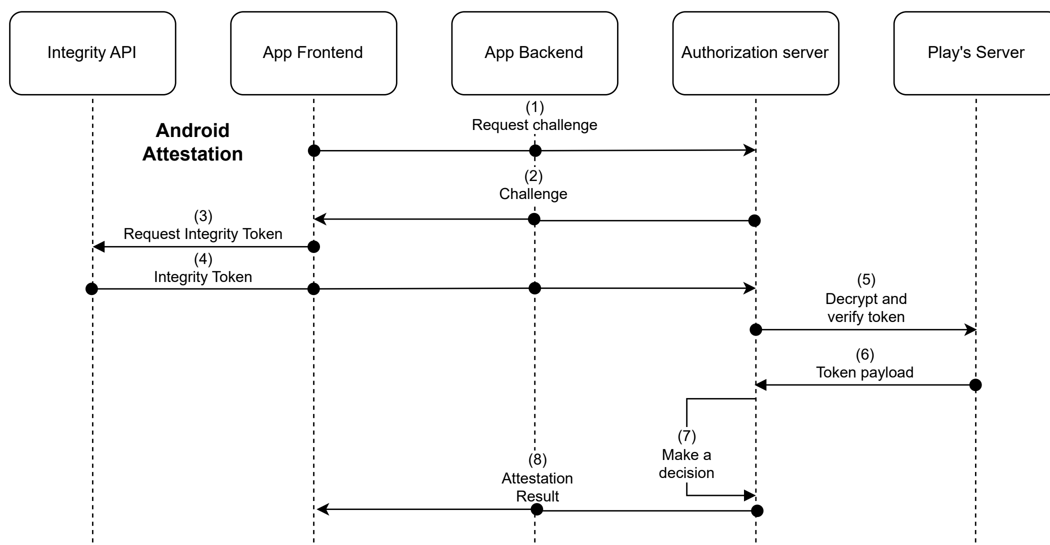

Figure 2.5: The flow of Attestation in Android

(1) The app frontend requests a challenge from the authorization server.

(2) The authorization responds with a challenge, which is a cryptographic nonce.

(3) The app frontend creates a hash of all request parameters, including the nonce, and sends it to the Integrity API requesting an integrity token.

(4) The API responds with an integrity token sent to the authorization server via the app. This response is signed with a key pair stored in the secure element. This key pair, in turn, is signed by a certificate chain that leads up to a Google root certificate.

(5) The AS requests the Google Play server to decrypt and verify the token. It verifies that the key used to sign the challenge response chains up to a Google root certificate.

(6) The Play server sends the token payload back to the AS. This contains information about the app, the device the app is installed on, and the hashed version of the initial nonce.

(7) The AS makes a decision based on all the information it has received.

(8) The AS delivers its decision to the app back- and frontend.

2.3.3 iOS Attestation

Similarly to Android, iOS also uses a secure element to store hardware-backed keys that can be accessed via APIs. A new one is generated if a key pair dedicated to that particular application is not present in the secure element at the beginning of the attestation. If a key pair already exists, it is be verified and used if it is valid. The client receives a key ID to access the key. The client then sends the challenge it obtained from the authorization server to the attestation system, which responds with some attestation data. This data contains a certificate chain and signature data. Verifying this signature is performed by walking up the certificate chain; if it is valid, it ends in Apple's root certificate. If this verification succeeds, additional checks surrounding the hardware and software is be performed. To be able to use iOS attestation, one would need to have a phone that can support iOS 14 [4], which is compatible with iPhone 6s and later [2].

Figure 2.6 and the description below explain in detail how attestation in iOS is performed step by step.

(1) The app frontend requests a challenge from the authorization server.

(2) The authorization server responds with a challenge, which is a cryptographic nonce.

(3) The app frontend makes a request to the Device Check API located on the device to create a new key pair. If a key pair exists for that particular client, those keys are validated. If there exists no key pair for this client, the API generates a new key pair using the secure enclave on the device and store the keys in the secure enclave. Apple's root certificate then sign the private key generated.

(4) The Device Check API responds with a key ID linked to the private key back. This ID needs to be stored by the app for future use.

(5) The app frontend makes an attestation request to the Device Check API that

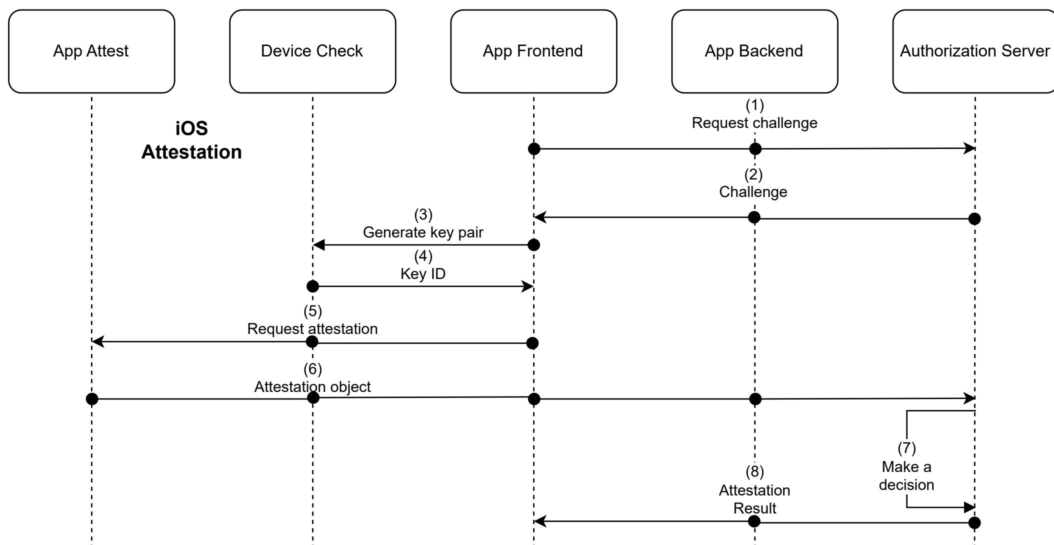

Figure 2.6: The flow of Attestation in iOS

sends that request to the App Attestation service via the network. This request contains a hash of the account ID together with the challenge from the authorization server and the key ID received from the previous step.

(6) The App Attestation server hosted by Apple verifies the request, and if the verification succeeds, it returns anonymous attestation data. This data consists of a list of certificates signed by Apple, Authenticator data, and a risk metric receipt. This data is then sent to the authorization server for verification.

(7) The authorization server verifies the attestation data. This is done by walking through the certificate chain in the certificate list and verifying that it leads up to Apple's root certificate. Also, the leaf certificate contains the hash constructed in Step 5. The server reconstructs this hash and compares it to the one in the certificate to protect against tampering. The Authenticator data contains a hash of the app identity that the server could use to verify the app's identity. The risk metric receipt is used to check the number of hardware-backed keys generated by a device, which is used to detect fraudulent behaviour. Depending on the results of these checks, the server decides if this app and device is legitimate.

(8) The authorization server indicates to the app frontend the result of the attestation process.

2.4 Multi-Factor Authentication

Multi-factor authentication (MFA) requires the user to use more than one authentication method [35]. This could mean that after the user has typed in their credentials, they would also have to input a code sent to their email or retrieve a code from some authenticator device. This provides an extra layer of security and ensures that even an attacker who has managed to steal the username and password is not be able to log in to the account.

How often the user would have to provide these multiple factors of authentication is up to the developer. It could be that every time the user logs in with their credentials, or it could be with every new device, they have to do it once. The security benefits of multiple authentication factors and how the user experience is affected need to be taken into account and weighed against each other.

2.5 OAuth 2.0 Best Current Security Practices

BCPs are very important to follow in order to keep applications secure. As studied in the most recent RFC draft regarding BCP in OAuth 2.0 [32], there are several scenarios to consider when maintaining BCP but also update those practices depending on the evolving landscapes of OAuth. The practices mentioned also apply to OAuth 2.1.

This draft also shows that these practices need to evolve and adapt as OAuth implementations are attacked using known weaknesses. OAuth is also used in more dynamic scenarios than originally considered, such as the relationship between the client and the authorization server. Technology has changed since OAuth was created, such as threats when redirecting browser requests. OAuth is also used today in environments with higher security demands than was originally anticipated, e.g., Open banking. All of these reasons show the importance of following the BCP.

In the following subsections is the BCP as considered by the OAuth working group.

2.5.1 Protecting Redirect-Based Flows

There are four main guidelines to follow for redirect-based flows. When performing the comparison of client redirect URIs against pre-registered URIs, authorization servers must use exact string matching except for port numbers in localhost redirection URIs of native apps. This can help prevent leakage of access tokens and authorization codes.

Clients and authorization servers also must not expose the open redirectors to prevent the exfiltration of access tokens and authorization codes. Open redirectors are URLs collected from a query parameter that forwards the user's browser to capricious URIs.

Clients must also prevent cross-site request forgery (CSRF) attacks. CSRF attacks are requests sent to the redirection endpoint that a malicious third party sends and not the authorization server. This protection can be achieved by either using the built-in CSRF protection in PKCE, by using OIDC as the nonce parameter that provides CSRF protection, or using one-time use CSRF tokens.

If a client is communicating with several authorization servers, a defence against mix-up attacks is required. To achieve this, clients should use either the "iss" parameter according to [46] or use an alternative based on an "iss" value from the authorization response. If none of these options are available, clients may use distinct URIs instead to identify the token endpoints and authorization endpoints. If an authorization server is redirecting a request that could contain user credentials, it must avoid forwarding these credentials.

The redirected-based flow that is relevant for us is the authorization code flow. The most important practice in this area is that the clients must use PKCE, and the authorization servers must support PKCE. More specifically, the PKCE challenge must be specific to that transaction and securely bound to the user agent and client that the transaction started. Also, the PKCE code challenge methods used by the clients should not expose the PKCE verifier in the authorization request.

All of these practices are to ensure that the security provided by PKCE is maintained. It also prevents authorization code injection attacks. Another example of an attack would be a PKCE Downgrade Attack, which the authorization server can mitigate by ensuring that the token request with a verifier parameter is only accepted if the challenge parameter is present in the authorization request.

2.5.2 Token Replay Prevention

For access tokens, a client would need to be able to prove that they possess a particular secret in order for the resource server to be able to accept that token. An access token is constrained to a particular client with their secret. This is to prevent the misuse of leaked or stolen tokens. To achieve this, authorization and resource servers should implement mechanisms to prove that the client possesses this secret. This could be done by either using Mutual TLS for OAuth 2.0 [15] or using OAuth 2.0 DPoP.

For refresh tokens, if using a public client, then the token needs to be constrained to the user similarly to access tokens, or the refresh tokens would need to be rotated. For confidential clients, refresh tokens can only be used by the client for which the token was issued.

2.5.3 Access Token Privilege Protection

When considering the privileges of access tokens, one needs to follow the principle of least privilege. An access token should be restricted to the minimum required

privilege for a particular use case. This could include the access of resources at a resource server. An access token should only be able to access the minimum amount of resources that satisfy the task for which the token was issued. An access token should also be restricted in the number of resource servers it can access and not be able to access resource servers that host resources that are not relevant to that particular access token. Following these principles mitigates the risk of access token leakage and also the potential risk for a client to access resources that they are not authorized to access.

2.5.4 Client Authentication

Regarding client authentication, authorization servers are recommended to support this if possible. The suggested method is to use asymmetric (public key) methods to achieve this. Such authentication methods include using MTLS or DPoP. By using asymmetric methods, the authorization servers do not need to store sensitive symmetric keys such as client secrets, and as such, they are more secure.

2.5.5 Other Recommendations

These recommendations are more general and can influence the security of different parts of the system using OAuth. The first thing is that clients should not be allowed by the authorization server to modify their client IDs as this could introduce confusion at a genuine resource owner, e.g., if a user wants to log into company A's app. However, the app says it is company B's app during the process. When it comes to communication between clients and servers, it is recommended to use end-to-end TLS. In particular, authorization responses must not be transmitted over unencrypted network connections.

2.6 FAPI 2.0 Security Profile

The Financial-grade API security profile, or FAPI for short, is a security profile for APIs that is based on the OAuth 2.0 Authorization Framework and other specifications with the purpose of protecting APIs that are in a high-security situation [24]. This profile was originally developed for the financial sector, but the design can also be applied to protect APIs in other high-security situations using OAuth. The following points in this section present the different layers that need to be considered when implementing FAPI.

2.6.1 Network Layer Protections

To protect the network layer, all the connections between authorization servers, resource servers, and clients need to be made through TLS. To also protect these TLS connections from network attackers, all endpoints need to use TLS version 1.2 or later, and the connections should follow the recommendations in Secure Use of Transport Layer Security [47]. DNSSEC should be used to prevent DNS spoofing attacks, and all TLS servers should perform certificate checks. All TLS connections,

both from servers and clients, should also only use the cipher suites specified in the FAPI RFC.

2.6.2 Requirements of The Actors

There are different requirements placed upon the different actors of the OAuth flow. These requirements are general and apply to different kinds of applications using OAuth. The authorization server has several requirements placed upon it. It should reject the resource owner password credentials flow and only issue sender-constrained tokens with either MTLS or DPoP. For clients, they should use sender-constrained access tokens with either MTLS or DPoP [24]. The authentication method supported should be either MTLS or "private key jwt".

2.6.3 Cryptography and Secrets

The FAPI security profile states that the participants in the OAuth flow should follow the BCP regarding JWTs during the creation and processing of JWTs. The algorithm to use should only be selected from a set of predetermined algorithms, and the "none" algorithm should not be accepted. For the different keys, RSA keys should have a minimum length of 2048 bits, and for elliptic curve keys, the minimum length should be 160 bits. Also, credentials such as authorization codes, access tokens, and refresh tokens should be created with at least 128 random bits to mitigate brute-force attacks on the credentials.

2.6.4 Security Considerations

Different types of attacks exist for different aspects of the OAuth protocol. The FAPI specification lists different attack vectors and measures that must be considered during implementation.

Firstly, attacks on access tokens and refresh tokens should be considered. It is worth considering using the short-lived access token to reduce the time window for an attack. Refresh tokens could allow clients to rotate their constrained keys to maintain good security hygiene. Depending on the length of the grant, short-lived access tokens together with refresh tokens could be a good solution. One thing to keep in mind is the trade-off between resilience and performance regarding the lifetime of tokens. Depending on the lifetime, it could affect the authorization servers.

Another thing to consider is replay attacks of DPoP proofs, where an attacker could replay the request with the corresponding DPoP proof. Possible mitigations to this include using short-lived nonces in the proof to decrease the time window for replaying. Message signing or MTLS are also solutions to this problem. The choice of mitigation strategy should be considered, along with the potential trade-offs of each strategy.

Another aspect to consider is using JSON Web Key Sets (JWKS) URI endpoints, which could be used to distribute public keys. This is used when clients and au-

thorization servers want to verify payloads signed by other parties with their corresponding private keys.

2.6.5 FAPI CIBA

The FAPI CIBA profile is a security profile defined by the OpenID foundation that specifies the BCP in accordance with the FAPI standard for clients that use the CIBA flow [49]. The security profiles include practices for designing and implementing authorization servers and confidential clients. It also specifies different vectors at which an application could be attacked. One is the scenario where a malicious actor wants to start an authentication session of a legitimate user without its knowledge or consent. This can be mitigated by either having the "login hint" parameter have the properties of a nonce or using a QR code to initiate the flow. Another interesting vector is if a malicious actor tries to start an authentication process for a user while the user starts the flow. If two actors try to initiate a flow simultaneously, this could be mitigated by issuing a timeout for that user.

2.7 OAuth 2.0 Threat Model

OAuth 2.0 has its own threat model with corresponding RFC [31]. This document analyzes threats directed toward the OAuth protocol and lists countermeasures to each threat. Our analysis is based on this RFC when looking at threats and countermeasures to our protocol. The most relevant threats against our protocol are listed in this section.

2.7.1 Assumptions About Attacker

The attacker is assumed to have unlimited resources when conducting its attack. They can also eavesdrop on all encrypted traffic between the client, the authorization server, and the client and the resource server [31]. For a protocol to be considered secure, it must work despite the attacker having such power.

2.7.2 Attacker Obtains The Refresh Token

If the attacker obtains the refresh token, they could use it to get new access tokens and essentially have access to the resources until the refresh token is revoked or expires. To mitigate this, there are several things the system can do to limit or stop the impact. Having the refresh token be bound to the app's client ID and then having the AS check if it matches with every incoming request helps. The AS can also limit the token's scope, so even if the attacker gets access to the resources, the damage they can do is limited. Revoking the refresh token if the AS suspects it has been compromised is also possible. Specifically for mobile apps, the token must be stored in secure storage so that even if the attacker manages to read memory for the app, it cannot steal the token.

2.7.3 Attacker Obtains Access Token

If the attacker gets the access token, they query the resource server directly for resources. To mitigate this, it is essential to store the access token securely. The principle of least privilege applies here by only giving the token access to precisely what it needs and nothing more. Finally, limiting the token lifetime is recommended because even if the attacker gets the token, they cannot use it for long.

2.7.4 Attacker Obtains Authorization Code

To stop the attacker from using a stolen authorization code, the code must be bound to the correct client. The AS can then ask the client to authenticate itself when sending the authorization code. Similar to access tokens, the lifetime needs to be kept short. The AS can also keep track of how many times it has been sent a specific authorization code, reject the code the second time it gets it, and revoke tokens granted based on the first time it got the code.

3
Related Work

This chapter looks into already existing solutions that claim to solve a similar problem as this thesis. There exists a very scarce amount of research about solutions that use OIDC/OAuth 2.1 for user authentication and authorization in native mobile apps without the use of redirection. This chapter highlights one. Also, relevant current ongoing research regarding client authentication and first-party applications for OAuth 2.0 is presented.

3.1 HAAPI

One proposed solution is the Hypermedia Authentication API or HAAPI [17], which is an API created by Curity that mobile clients can use to authenticate themselves without requiring a browser to perform the authentication and consent interactions. This can be achieved by first attesting and authenticating the client and then performing the authorization code flow with PKCE but without browser redirects using this API endpoint. However, this solution is not an industry standard, and no RFC is accepted. The flow of this solution can be seen in Figure 3.1.

In this protocol, the terms "app" and "client" are used synonymously. The steps of the protocol flow are described in more detail below. The attestation procedure is the same as described in Chapter 2.

(1) The user initiates the flow by clicking on the login link in the app.

(2) The app requests a challenge from the authorization server in order to obtain a client attestation token (CAT). This request is sent to a specific endpoint designed for this purpose with the client ID as a parameter. When performing this request, the app is not authenticated but only identified through the client id.

(3) The authorization server checks if it can identify the app through the provided client ID and if the client can use HAAPI. If that is the case, the authorization server sends a challenge to the client.

(4) The app sends a request for attestation to the attestation system, which is usually implemented in the mobile phone by the manufacturer. In this request, the app sends the challenge it received in the previous step to the attestation

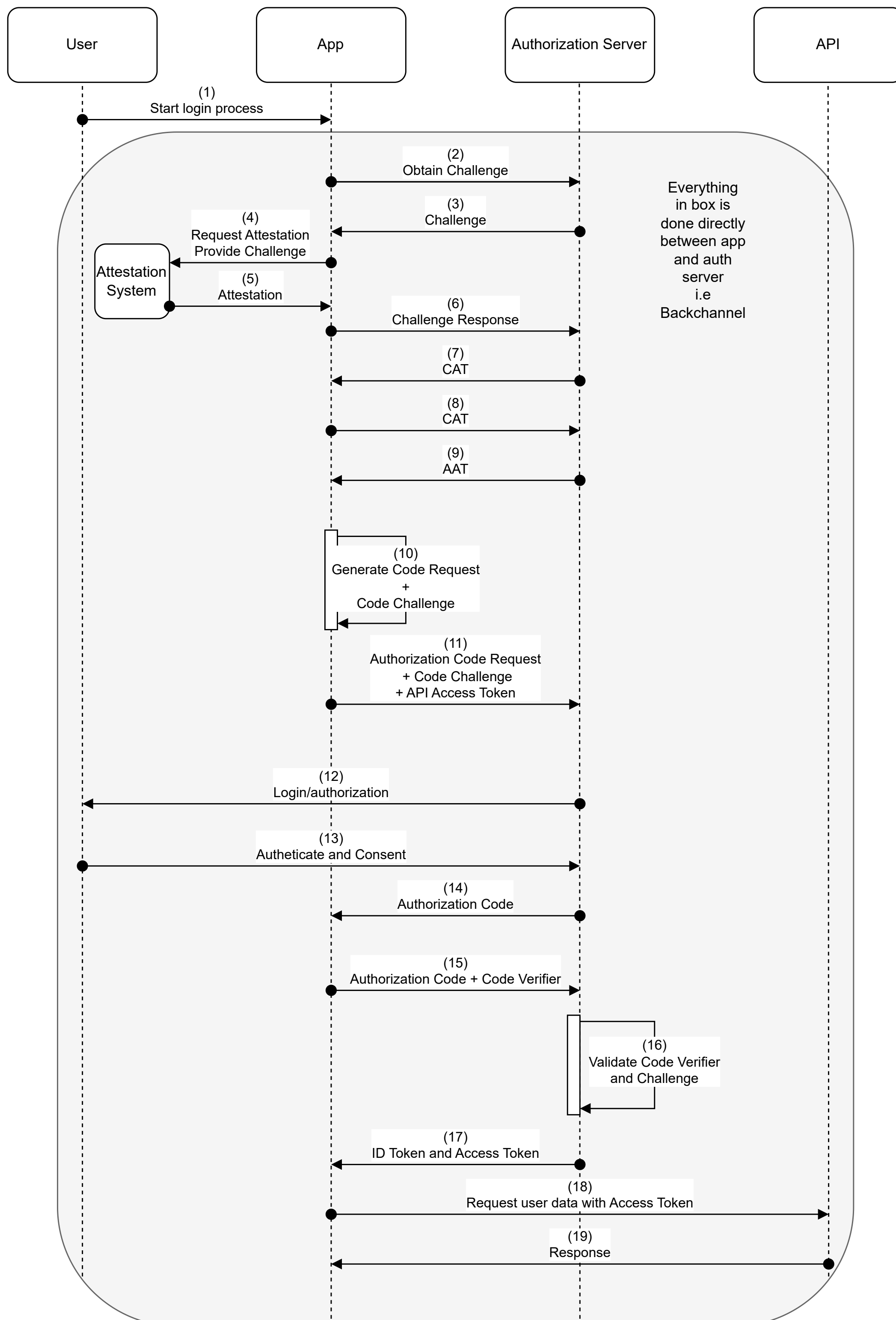

Figure 3.1: The flow of HAAPI

system.

(5) The attestation system performs the attestation. This is done by taking the identifier of the app, the hash of the challenge, and the environment data (e.g., if the app is run in production or if the key used for the signature is stored in hardware or software) and signs it using the private key stored in the secure element of the mobile phone. The app identifier consists of the app package name and the signing key digest. This signed response is sent back to the client.

(6) The app sends the challenge response to the authorization server. This response includes the signed attestation obtained in step 5, as well as the corresponding public key to the private key that was used to sign the response. The authorization server then verifies this challenge-response by checking if the certificate used to compute the hash is valid. This is done by walking through the chain of certificates all the way up to Google's root certificate. This works because the chain is part of the signature and is cryptographically bound to it.

(7) When the authorization server has verified the chain of certificates, it sends a CAT to the client.

(8) The app sends the CAT to the HAAPI endpoint of the authorization server when it wants to authenticate itself to HAAPI in order to obtain an API Access Token (AAT). When sending the CAT, a proof-of-possession of the private key used to sign the challenge response is required. This is done by verifying the signature of the token by walking through the chain of certificates all the way up to Google's certificate. Then, the CAT is used to authenticate the client by using the CAT as authentication details in the regular client credentials flow, which outputs an AAT.

(9) The authorization server sends the AAT to the client.

(10) The app generates a Code verifier and a Code Challenge for PKCE.

(11) An authorization code request, the code challenge, a callback URI, and a scope of what data the app wants to get access to are sent to the authorization server. This is sent to the HAAPI endpoint together with the AAT in order to show that the client can access this API.

(12)(19) Same as in the regular authorization code flow with PKCE depicted in Figure 2.1 except that the communication between the client and the authorization server is through the HAAPI endpoint and all communication is done through backchannel communication. If the client is of first-party origin, the username and password is entered directly into the app. If the client is of third-party origin, the method of providing the credentials varies. Suppose a username and password provided by the user is not sufficient to authenticate a user, that is multiple factors are required. In that case, the credentials is

inputted directly into the app as well, and the other authentication factor is performed out-of-band with SMS one-time-password or BankID. If a username and password are enough to authenticate the user, then a method that does not collect these credentials directly is required. This can be done using the previously mentioned out-of-band authentication methods.

The CAT contains information about the expiration time, OS used, app client ID, a key to verify the authenticity of the CAT, and so on. The CAT is bound to the client that passed the challenge to ensure that even if intercepted, it cannot be used by the attacker. When the CAT is verified, everything about it is checked. The expiration time, that the key the signed the challenge matches the certificate, that the client is registered and should have access to the API, and that the CAT has reached the correct endpoint, among others [16].

3.2 OAuth 2.0 Attestation-Based Client Authentication

OAuth 2.0 Attestation-Based Client Authentication is an RFC draft published by the Network Working Group that specifies a method for public clients to authenticate themselves to an authorization server by using attestation [33]. A client does this by including an assertion token in a request to the authorization server, which is then bound to a public key and proven possession of with DPoP. The flow of these authentication mechanisms can be seen in Figure 3.2 and is described step by step below.

(1) The client frontend generates a new public/private key pair only used for this session and optionally other attestation data.

(2) The client frontend requests a client attestation JWT from the client backend by sending the newly generated instance key and, optionally, other attestation data.

(3) The client backend validates the key and the optional data and generates an attestation JWT that is cryptographically bound to the instance key.

(4) The client backend responds to the client frontend with the client attestation JWT.

(5) The client frontend generates a proof of possession for the client attestation JWT.

(6) The client frontend makes a request (e.g., token request) to the authorization server, which includes the proof and the attestation JWT. The authorization server validates the attestation JWT and the proof.

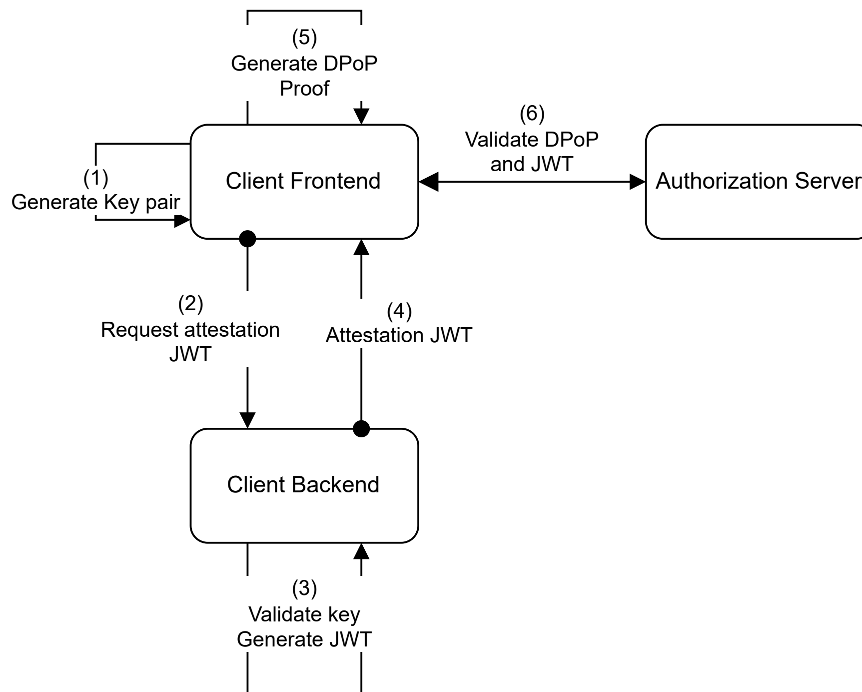

Figure 3.2: OAuth 2.0 Attestation Based Client Authentication

3.3 OAuth 2.0 for First-Party Applications

OAuth 2.0 for First-Party Applications is an RFC draft published by the Web Authorization Protocol workgroup that specifies an authorization code flow with a more native experience [41]. Since this protocol requires a higher degree of trust between the acting parties, it is primarily designed for first-party applications. This native experience is achieved by having the client application communicate directly with the authorization server, which in turn can respond with error messages in order to obtain more authorization information until it can authorize a user. The flow of this can be observed in Figure 3.3, and a detailed step-by-step description can be found below.

(1) The user initiates the login flow by either pressing a login link or providing some login information.

(2a) The client application makes an authorization request to the authorization server, optionally with some login information retrieved from the user.

(3a) The authorization server determines if the information in the request provided by the client application is sufficient to authorize this user. If not, then it responds with an error message to the client that says more information is required for authorization.

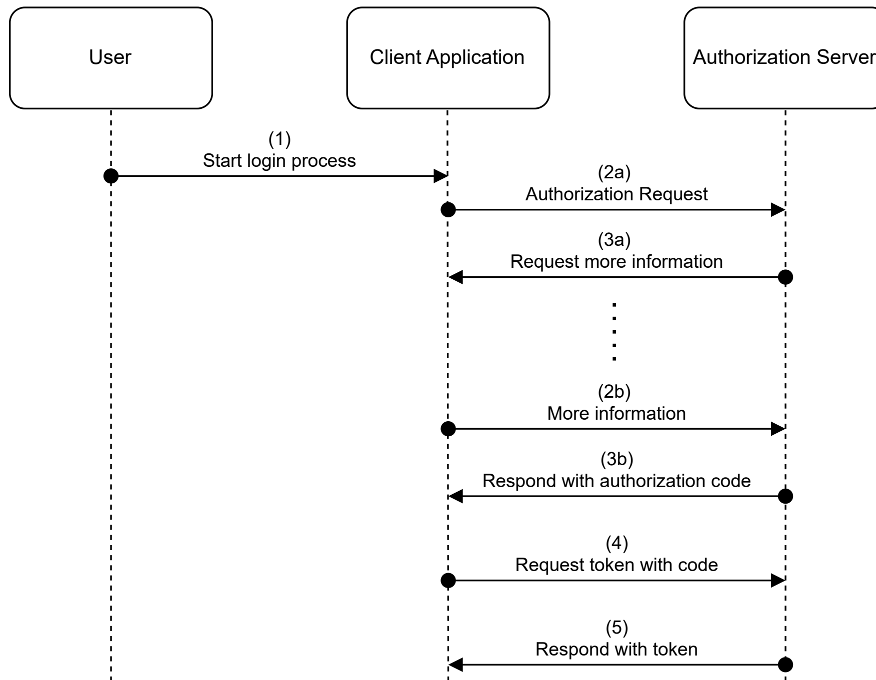

Figure 3.3: OAuth 2.0 for First-Party Applications

(2b) The client application provides more information as requested from step 3a.

(3b) The authorization server verifies the provided information and responds with an authorization code.

(4) The client application makes a token request using its authorization code.

(5) The authorization server responds with an access token.

3.4 User Experience

Several studies have been conducted on the user experience and user perception of different authentication schemes. A study done in Germany found that even though good passwords can be hard to use due to them being difficult to remember, people still really like to use passwords [52]. This is probably because people are really used to using passwords. They also found that using fingerprinting in the authentication process was highly preferred by users and was even perceived to be a more secure option among those tested.

4
Our Approach

This chapter describes our approach and the work process of this thesis and also describes the tools used to conduct the research, particularly in the app analysis.

4.1 Pre-Study

The methodology in this thesis consists of first performing a comprehensive study of OAuth 2.1 and OIDC. This includes the basic concepts regarding these protocols, as well as the different flows contained within the protocol and the different properties of these flows. This is done in order to get an understanding of how the protocol currently works and to identify the critical parts of the existing protocol that we want to change with our protocol.

Next, we analyse one of the few existing solutions to this problem, HAAPI, in order to get an understanding of how this solution works. We also focused on which parts of this solution are important to our problem and that we can use in our solution.

4.2 App Analysis

We conduct an app analysis of how existing apps perform the login process for users. For this study, we use the program Burp Suite Community Edition, which is software used to perform different security tests on web pages and mobile apps [43]. We use Burp Suite to view the HTTPS messages sent between the app and the servers during the login procedure.

More specifically, we use the Burp Proxy Listener to listen in to the traffic between the app and the servers. We configure the proxy to listen in on port 8082 and made sure that the computer with the proxy and the phone are on the same network [44].

The phone we use during testing is an iPhone 12 Mini with iOS version 16.5. On the phone, we install a special certificate from Burp to be able to decode and read the traffic between the apps and the servers during the login process [44].

The apps studied were chosen for different reasons, the main one being that they all seemed to handle the login process natively; that is, it was performed without leaving the app. They also do not use certificate pinning, which is a security mechanism

used to authenticate the TLS connection between the client and server [50]. This enables us to read the traffic sent between the client and the server in cleartext by installing our own certificate.

4.3 Protocol Design

We designed our protocol by considering what we found in the pre-study as well as the app analysis, especially what parts could be useful in our case, as well as other information that could be useful. Throughout the process, we came up with different versions of the protocol. For each version, we analyse what parts of the protocol works and what parts to improve in accordance with the BCPs for OAuth, FAPI, and the OAuth threat model mentioned in Chapter 2. This was done by going through each criterion of these RFCs and seeing if our protocol upholds it. We then modify the protocol according to that analysis. This process is repeated until we arrive at our final version of the protocol.

We wrote the pseudocode using Python-like code to represent the logical flow of the protocol. We identify which parts of the code that needs a reference to an already existing library and sought those libraries out. When choosing what libraries to use, we consider who developed the library and how widely used the library is in order to analyse the usability and security of these libraries.

Finally, the protocol as a whole and its accompanying pseudocode are taken into consideration when discussing and analysing how the user experience feels for our protocol. We consider the complexity of the protocol, that is, how easy it is to understand and implement, by discussing the protocol with security experts at Omegapoint. We also consider the UX when interacting with the app during the authentication process.

5
App Analysis

This chapter presents our findings from analysing the login flows of different apps. Due to our findings that some of the apps use flows that are not aligned with the BCPs, e.g. one flow is deprecated and will be removed in OAuth 2.1, we do not mention the companies behind these apps by name. We instead refer to them as app A, B, C, D, and E, respectively. The reason for analysing these apps was to give us inspiration when designing our protocol and also learn more about how login processes work in apps used today.

5.1 App A

App A uses the ROPC flow and makes a simple POST request to the server with the username and password directly. It then immediately gets an access token back in response. The access token also has a lifetime of one year and is a bearer token.

5.2 App B

App B uses the Authorization Code flow. The real tokens are not sent to the client; however, they are stored on a backend server. The client then receives reference tokens for both the access token and the refresh token that it can send to the backend server, which in turn can use the real tokens on the client's behalf, depending on the action.

5.3 App C

App C also uses the ROPC flow. It does, however, also send the client secret in the post request with the username and password to the authorization server. The server responds by sending an access token and a refresh token back to the client. The access token has a lifetime of 12 hours and is a bearer token.

5.4 App D

App E uses the Authorization Code flow with PKCE and OIDC, with the use of redirection via a browser. This is done in an embedded web view, however, and not in the web browser provided by the OS.

5.5 App E

App E makes a POST request with the credentials but also requires multi-factor authentication every time a user logs in. This is done by sending an email with a code to the user that the user, in turn, inputs to the app to authenticate itself.

5.6 Analysis

During our app analysis, we saw that the ROPC flow and the authorization code flow were primarily used. It was interesting to see that companies still use the ROPC flow, even though the BCPs highly recommend against its usage. The motivation behind its usage could be that the app is a first-party app, and the company thus believes it has fewer security demands so that they could use the simpler but more insecure flow.

One of the companies that used the authorization code flow used reference tokens instead of sending the access token directly to the frontend. We found this mechanism to be beneficial for our protocol since the real access token would only be stored on the backend server. It is also an extra layer of protection against token leakage since the real access token does not reach the frontend. A reference token also gives more control over the revocation since it is easier to revoke a reference token than an access token.

The app analysis connects to the first research question, namely to see how real in-use apps that have a native login experience without browser redirection handle the login process. It shows that it is possible to have such a protocol, but it also shows that it is difficult to follow all security standards since several of the apps do not follow all the BCPs. From our point of view, it also shows that a native app login without browser redirection is friendlier to the user compared to the regular code flow. This connects to the third research question regarding the user experience. The fact that companies also design their apps with a more native login experience shows that there is a demand for handling the login process natively, even at the cost of not following the BCPs.

6

mAuth - Our Proposed Protocol

This chapter presents what our proposed protocol looks like. This section only states the result of this thesis, and the motivation regarding our design choices and other analysis are provided in Section § 7.

6.1 Detailed Description

Below is the detailed description of our proposed protocol, which is called mobile authentication (mAuth). Figure 6.1 depicts the flow of mAuth. The protocol is based on OAuth, described in Section § 2.1.

(1) The user initiates the authentication/authorization process, which could be done by, e.g., clicking a login button or a link.

(2) The client frontend sends an attestation challenge request to the authorization server that is forwarded via the backend server. This request includes the client ID, which identifies the client at the authorization server. The identification is used to show the AS whom the client is claiming to be.

(3) The authorization server responds with an attestation challenge to the client via the backend server. This challenge is a cryptographic nonce and is used to prevent replay attacks.

(4) The client frontend generates a new client instance key pair that are used during the DPoP process, and this key pair is only used during this session. The private key is stored securely on the device as described in Section § 2.3.1.

(5) The client frontend makes an attestation request to the attestation system of the mobile device. This request includes information about the application, the operating system of the device, the execution environment, and the public key generated in the previous step. How the attestation system works in detail is described in Section § 2.3.

(6) The attestation system verifies and signs the provided information from the client. This process varies depending on whether an Android or iOS device is used, but the results are similar.

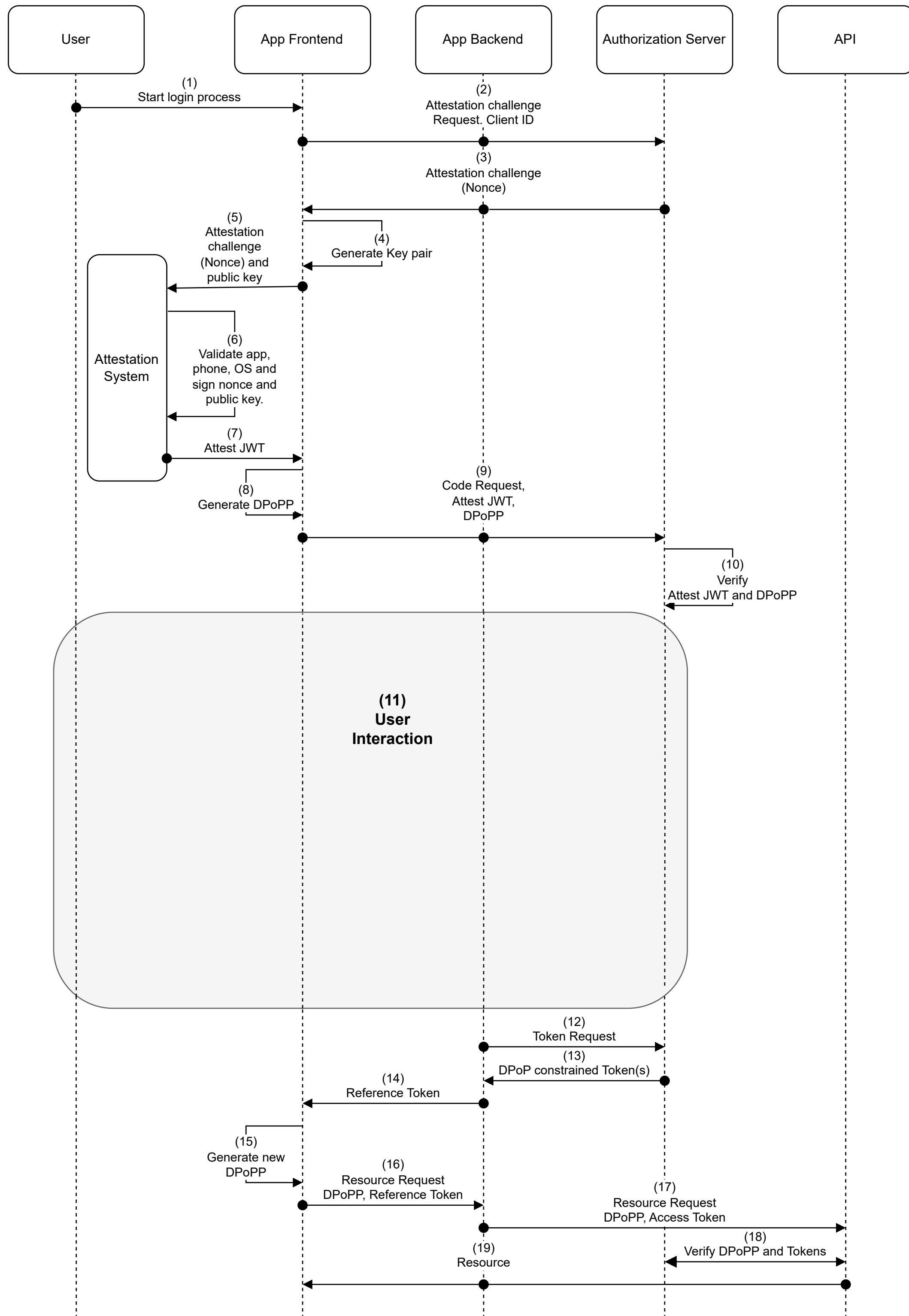

Figure 6.1: The flow of mAuth

(7) The attestation system sends a signed attestation JWT back to the client frontend. This JWT contains information about the result of the verification, the nonce, and the public key generated in step 4.

(8) The client frontend generates a unique DPoP Proof (DPoPP) JWT, as described in Section § 2.1.6, that proves that the client is in possession of the private key generated in step 4.

(9) The client makes an authorization code request to the authorization server via the backend server. In this request, it attaches the attestation JWT and adds the DPoPP in the HTTP header.

(10) The authorization server verifies the JWT and DPoPP and, if approved, sends a message back to the client via the backend server that says that the client has been attested and needs to provide login information.

(11) Here, one of the three different flows described in Subsections 6.2, 6.3, and 6.4 can be used depending on user preference. The three flows are explained later. For now, it is enough to know that at the end of this step, the app backend gets an authorization code that it can use for the token exchange.

(12) The backend server makes a token request by sending the authorization code to the authorization server. This authorization code is not sender-constrained as this communication is only between the backend server and the authorization server, i.e., backchannel communication.

(13) The authorization server responds with DPoP-constrained access/ID- and reference tokens and optionally a refresh token as well.

(14) The backend server sends the reference token to the client frontend, which the frontend can use to access the access/ID token or refresh token stored on the backend server.

(15) The client frontend generates a new unique DPoPP that proves that the client is still in possession of the private key generated in step 4.

(16) The client frontend makes a resource request to the backend server with the DPoPP JWT to retrieve some data from the resource server using the reference token.

(17) The backend changes the reference token for the actual access/ID token and sends it along with the rest of the request to the API.

(18) The API sends the tokens to the authorization server along with the DPoPP to verify them.

(19) The resource server responds with the requested data that is forwarded from the backend server to the client frontend.

Using refresh tokens also opens the possibility of rotating the keys used by the sender and constraining the tokens without the loss of the grant. This can be achieved by including a second DPoPP together with the old DPoPP when using the refresh token to get a new access token. The authorization server can confirm that this user can use this refresh token because of the current DPoPP, and it can also see which new key it needs to bind the following access token to with the second DPoPP.

MFA is also possible in our protocol. This is done similarly as described in Section § 3.3 by requesting more login information from another mechanism after receiving the login information from the first. In this way, one could combine several login methods in order to obtain stronger authentication. MFA could be used with either of the three login flows described in Sections 6.2, 6.3, and 6.4.

6.2 Integrated ROPC

While normal resource owner password credentials flow is deprecated in OAuth 2.1, our integrated version is designed to be a more secure way for first-party apps to still use it compared to using the version in OAuth 2.0. The integrated ROPC flow can be depicted in Figure 6.2, and below is a step-by-step description of the flow.

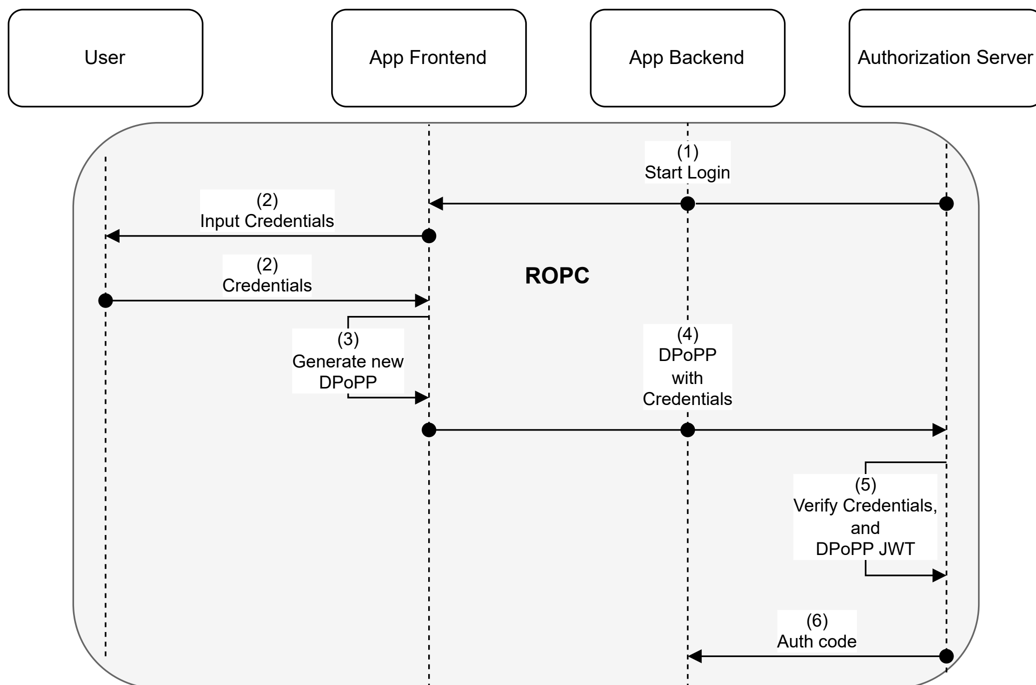

Figure 6.2: The flow of ROPC in mAuth

(1) The authorization server sends a message to the app frontend to begin the login process.

(2) The app frontend prompts the user for credentials. The user provides them.

(3) The app frontend generates a new DPOPP JWT with the credentials.

(4) The DPoPP JWT with the credentials is sent to the AS.

(5) The AS verifies the DPoPP JWT and the credentials.

(6) If the verification is successful, then the AS sends an authorization code to the backend server.

6.3 Integrated CIBA

Figure 6.3 illustrates the integrated CIBA flow used in our protocol, and below follows a detailed step-by-step description of the flow.

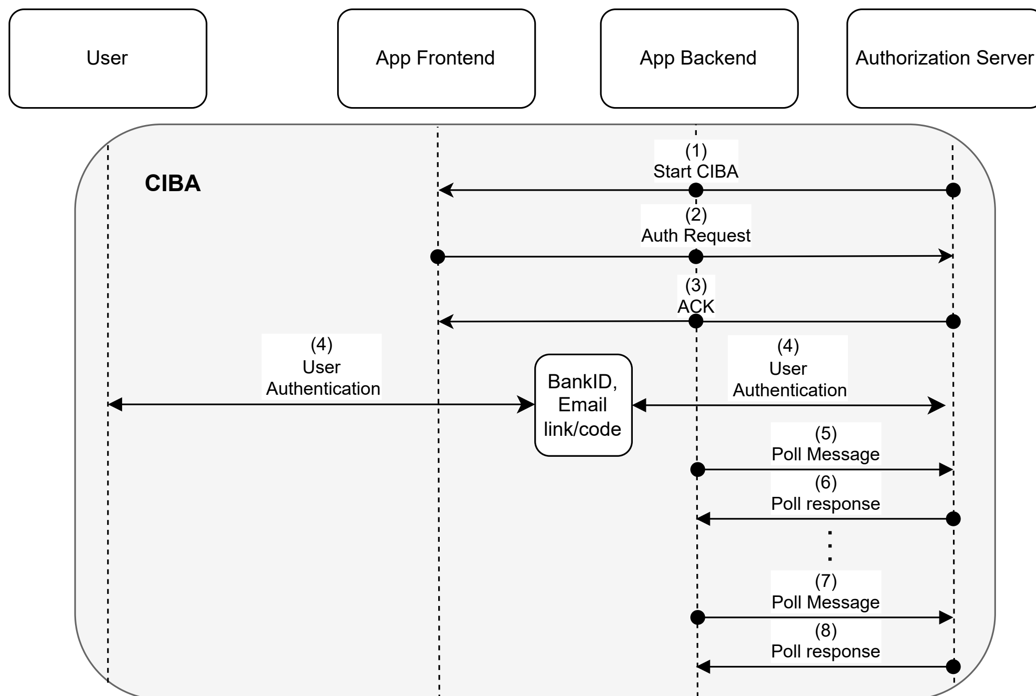

Figure 6.3: The flow of CIBA in mAuth

(1) The AS tells the client frontend to start a CIBA flow.

(2) The client frontend makes an authentication request to the Backchannel Authentication Endpoint at the authorization server.

(3) The authorization server responds with an ACK message that includes a unique identifier for that request and a parameter indicating the maximum lifetime of the transaction.

(4) The authorization server performs the user authentication by using an out-of-band authentication method, such as a mobile phone, to interact with the user, most often installed on the same phone as the client. The authorization server chooses the authentication method used depending on the situation.

(5) The backend server starts polling the authorization server at regular intervals, checking if the authentication process is done.

(6) The AS responds to the backend server that the authentication process is not completed yet.

(7) The backend server keeps polling the AS.

(8) When the authentication process is done, the AS responds with the relevant information to the backend server, which in this case is an authorization code.

6.4 Integrated FIDO2

In our integrated FIDO2 flow, the relying party is the authorization server and the FIDO2 client is the app front- and backend. The integrated FIDO2 flow can be observed in Figure 6.4 and a description of the flow step by step is described as the following:

(1) The authorization server sends a challenge to the client frontend. This challenge is a cryptographic nonce. The AS also sends a "get" request to indicate that it wants to authenticate a user.

(2) The app frontend proxies and sends the request and the challenge to the authenticator. In the request, the app frontend also includes the origin information of the request.

(3) The authenticator prompts the user, asking if the user wants to authenticate to the requesting AS. The user provides their credentials to the authenticator. This could be, e.g., a PIN or biometric credentials.

(4) The authenticator creates a response for the AS, which includes the origin info and the challenge. The authenticator then signs the response using its private key and sends the response to the AS, which is proxied via the app.

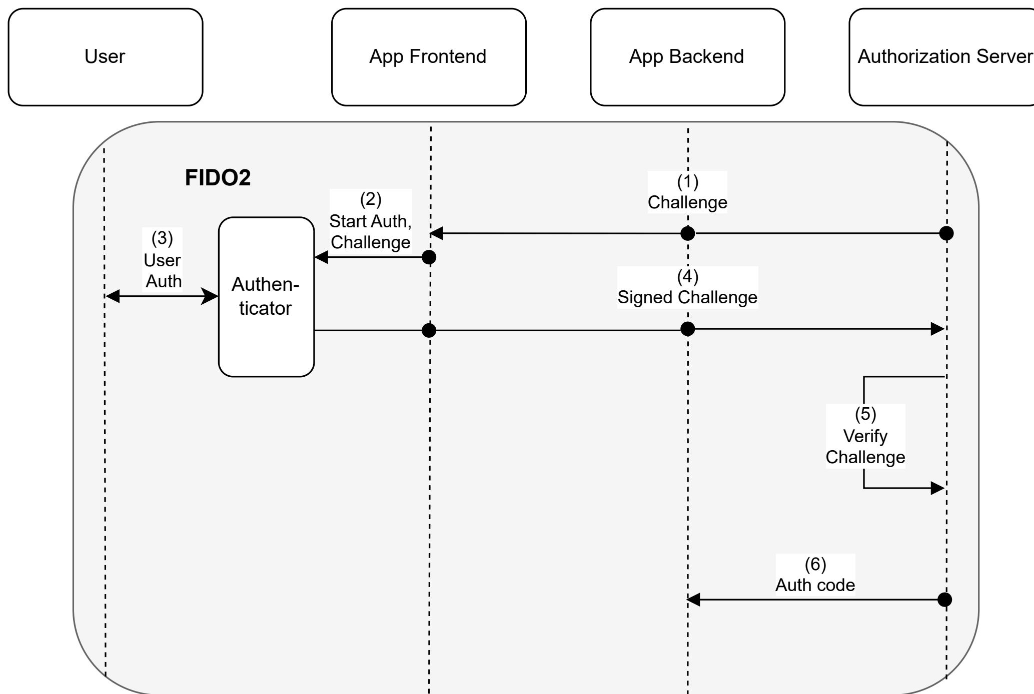

Figure 6.4: The flow of FIDO2 in mAuth

(5) The AS verifies the response from the authenticator. This includes validating the authenticity of the authenticator, the integrity of the response, as well as that the origin is the same as expected. Validating the origin prevents phishing attacks.

(6) The AS indicates to the app, and in turn, the user, the result of the authentication ceremony.

6.5 Client Verification for User

The attestation used previously is used by the organisation that created the application to verify that it actually is their app that is being used. It does not show the user that the app the user installed is legitimate. In order to remedy this, we have designed the flow in Figure 6.5.

(1) The user asks the client for its certificate.

(2) The client provides its certificate.

(3) The user opens up a web browser on the same device that the app is installed on, goes to the organisation that owns the app's website, and provides the certificate to the website.

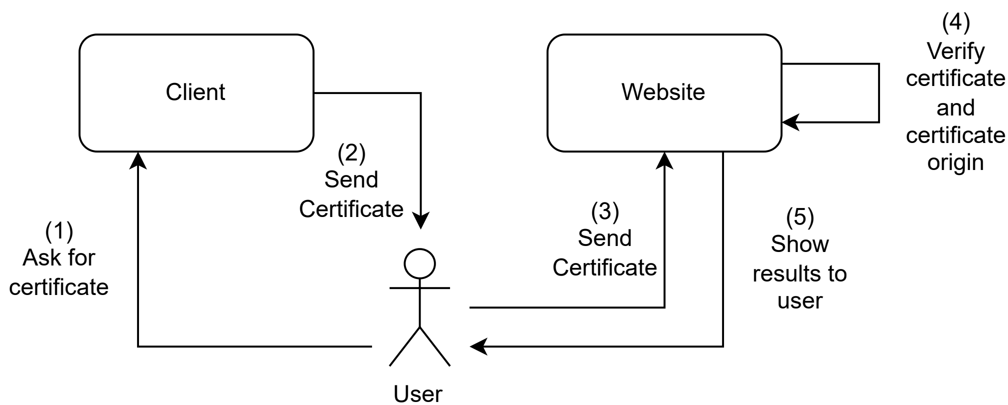

Figure 6.5: The flow for the user to verify the client

(4) The website verifies that the certificate is coming from a legitimate app and that the device the certificate claims the app is running on is the same as the browser is running on.

(5) The website shows the result to the user.

6.6 Security Analysis

When performing the security analysis of mAuth, it showed that the protocol depicted in Figure 6.1 followed most of the practices specified in the OAuth BCP, the FAPI Security Profile, the FAPI CIBA profile, and the OAuth Threat Model profile, described in Sections 2.5, 2.6, 2.6.5 and 2.7 respectively.

Always using TLS, having every token being sender constrained with DPoP, and performing an attestation of the app and its environment makes the protocol follow most of the practices. We also saw that following the principle of least privilege was beneficial in minimising the damage a token leakage would cause. Also, not using a browser eliminated several vulnerabilities, e.g., redirect URIs and CSRF attacks.

Many of the requirements specified in these RFCs were implementation details, which were considered to be out of scope for this thesis. There are some, however, that are not followed for a multitude of reasons, which are discussed in Section § 7.2. Arguments discussing the correctness of the protocol can be found in Chapter 7. To read the full security analysis of mAuth, see Appendix A.

6.7 mAuth in Pseudocode

We have written Python-like pseudocode to represent the logical flow of mAuth. The pseudocode can be observed in Appendix B. We have also gathered information about libraries that could be used to do a real implementation of this pseudocode.

Some of these libraries are a part of the standard library for their respective programming language, while others are libraries provided by companies.

For the generation of nonces, the standard libraries `java.security.SecureRandom`, `java.security.MessageDigest` and `java.util` can be used to achieve this [39] [40]. In this way, a random number can be generated that is then hashed and finally encoded into hexadecimal format. To generate a new client instance key pair for Android, the `java.security` package can also be used. For iOS devices, one can follow the official Apple developer documentation to achieve this [6]. To send HTTP requests, Golang has the `net/http` library built into their standard library [26]. This can be used in combination with the TLS library `OpenSSL` developed by Apache to provide HTTPS [38].

To create and validate JWTs, the `System.IdentityModel.Tokens.Jwt` can be used [36]. This library is owned and developed by Microsoft. For supporting CIBA, the `Duende IdentityServer` can be used for this case [20]. To provide support for DPoP, the `IdentityModel.OidcClient` can be used, which is also provided by Duende but is an open-source library [13]. For attestation, one can use the SDKs provided by Google and Apple and follow the official documentation of Play Integrity API or App Attest/Device Check to implement this functionality [19] [1] [7] [5]. For FIDO2, there exist several libraries that one could use. One such library is the `libfido2` library written in C and provided by Yubico [34].

6.8 Comparison with Regular OAuth Code Flow

Table 6.1 contains a comparison between our proposed protocol and the authorization code flow described in Section § 2.1.3, which is the most common OAuth flow used for native apps today. It compares different attributes of the regular code flow with the three different versions of our protocol.

Attribute	**OAuth**	**mAuth ROPC**	**mAuth CIBA**	**mAuth FIDO2**
Attestation	No	Yes	Yes	Yes
PKCE	Yes	No	No	No
DPoP	No[1]	Yes	Yes	Yes
Browser Redirect	Yes	No	No	No
Auth code Exchange	Yes	Yes	Yes	Yes
Client sees credentials	No	Yes	No	No
Bearer token usage	Yes[2]	No	No	No
User leaves the app	Yes	No	Yes	No

Table 6.1: Protocol comparison between our three protocol versions and the regular OAuth Code flow.

One major difference between regular OAuth Code flow and mAuth is the fact that there is no browser redirect present in mAuth, which was one of the goals of this thesis. Another difference is that the user leaves the app when using regular OAuth and when using the version of mAuth that uses CIBA. This does not happen when using the ROPC or FIDO2 version, as the user does not leave the app. One of the goals was to keep the user in the app, which is achieved by mAuth since there are several variants to use where the user remains in the app during the authentication/authorization process.

Another major difference is the fact that PKCE is not used in mAuth, while it is an integral part of regular OAuth. Instead, mAuth uses DPoP and sender-constrained tokens to achieve the same results as PKCE would have and more in the form of constraining the tokens to a specific user.

[1]It is possible to use in regular OAuth, but it is not part of the standard specification.

[2]In the standard version of OAuth Code flow, it specifies bearer token usage, but sender-constrained tokens are also possible.

7
Discussion

This chapter includes a discussion regarding the different parts of this thesis. This includes the analysis of our protocol regarding both the security and the pseudocode. We discuss what the user experience of mAuth looks like from our perspective. We also discuss what type of constraints exist regarding our protocol and potential future work.

7.1 Protocol Analysis

Our protocol accomplishes what we set out to do without being overly complex. A protocol could be extremely secure but very complex, which would make it hard to understand and implement correctly. This would hurt the overall security of the protocol. We believe that our protocol is not suffering from this issue as we tried to follow how different established and tested standards work and incorporate them together in our protocol.

The important mechanisms we use in our protocol are the attestation, the client instance key pair, the authorization code flow, and DPoP sender-constrained tokens. All of these mechanisms work together smoothly. The client instance key pair works nicely with DPoP as it creates fresh keys for the current session that constrain the usage of tokens to the correct sender. With the addition of attestation, it assures the authorization server that it can trust the client with this key pair. All of this is then integrated into the authorization code flow without disrupting the flow in any way, thus reaping the security benefits of this flow.

One major benefit that this protocol provides is the possibility of choosing which type of user interaction one would want to use in their app. The choice would depend on what type of requirements an app has placed upon it. All three of the user interaction versions are simple to incorporate with the rest of the protocol, as the inclusion of one version does not interfere with the rest of the protocol. This makes the protocol less complex and, in turn, more secure.

7.2 Security Analysis

This section includes a discussion of interesting points that came up during the security analysis. We also discuss the different BCPs that we choose not to follow.

7.2.1 ROPC

The OAuth BCP, OAuth Threat Model, and FAPI all state that the ROPC flow should not be used. In our case, we decided to include it in the protocol. However, we strongly recommend not to use this flow and instead use either the CIBA flow or the FIDO2 flow. The only use case where this flow should be used is in the case where the client application is of first-party nature, and the user wants to log in to their account that belongs to that same first-party organisation. We do not recommend using ROPC in the case of third-party applications since the risk with the ROPC flow is that the credentials are exposed to the client, which is terrible in this case.

One could consider using multi-factor authentication by having the authorization server after step 4 in Figure 6.2 request more login information from the user, which could be an OTP sent via, e.g. email or SMS. Then, the use of the ROPC flow could be motivated as the credentials exposed to the client are not enough to establish the identity of the user, as it also requires an OTP. However, the use of either the CIBA flow or the FIDO2 flow is still a better option as the authentication details are provided out-of-band with either, e.g. BankID for CIBA and facial recognition or fingerprint scanning for FIDO2. Also, the authentication details are not exposed to the client as in ROPC.

7.2.2 PKCE

PKCE is a BCP that is mentioned several times throughout the OAuth BCP RFC and OAuth Threat Model and is a very important part of the Authorization Code Flow. The main purpose of PKCE is to provide a mechanism for the authorization server to check that the entity that is requesting the token with the authorization code it issued is the same as the entity that initiated the flow. This is because the communication medium between the two phases of the flow is different, as obtaining the authorization code is done through frontchannel communication, and obtaining the access token is done through backchannel communication.

This is not an issue for our protocol. Firstly, the communication of the authorization code is always done through backchannel communication, as the code only reaches the app's backend and not the frontend. Secondly, with the use of attestation and issuing a challenge, the authorization server can make sure that it is communicating with a legitimate instance of the app. Finally, with the use of a client instance key and DPoP, the authorization server can make sure that it always communicates with the correct client throughout the authorization/authentication process. These mechanisms fulfill the same purpose as PKCE and more, which makes us believe that PKCE does not fill any function in our protocol and is, therefore, not used.

7.2.3 DPoP

While DPoP is a relatively new addition to the field of security, it offers many benefits for clients who are able to generate and store keys securely. Mobile apps

on modern smartphones are such clients. The extra layer of security it offers by binding the issued tokens to the key is nice as it makes stolen tokens useless to the attacker. It also makes it easy for the AS to know it is talking to the same client through the whole process and that the session has not been hijacked. This is built on the assumption that we can trust the secure storage on the phone. DPoP, in combination with attestation, makes the AS know that it is not only the same client throughout the session but also a legitimate client running on legitimate hardware.

7.2.4 Attestation

Because most phones in use today run either Android or iOS, and both Google and Apple provide their versions of attestation, it becomes a powerful tool we can always use. Being able to verify not only the authenticity of the app but also the operating system gives the AS the ability to tell the difference between malicious apps and real ones. While it only provides minimal additional security to the actual user of the app, it does provide a lot better security to the developer of both the app and the AS. This is also built on the assumption that we trust the service provided by Google and Apple.

The usage of this mechanism was inspired by HAAPI described in Section § 3.1. HAAPI is one of the very few proposed solutions to the problem of native authentication and authorization without browser redirects. Attestation is an important part of their solution that we thought would improve our protocol, and the fact that it is a mechanism provided by Google and Apple motivated our decision to incorporate it.

7.3 Pseudocode Analysis

In this section, we analyse our written pseudocode and the security of the libraries that could be used in a real implementation. The actual pseudocode can be found in Appendix B.

7.3.1 Complexity of Pseudocode

The pseudocode describes the logical flow of the protocol. Every step in the flow is described in the code, but many implementation details are left out, and it is up to the developer to choose how they want to implement it. We do have libraries we recommend to developers to use, but we have no hard rules on which ones should be used. It is worth mentioning that the client frontend and authorization server never talk directly to each other, as all communication between the two is sent via the backend server and then forwarded from there.

7.3.2 Library Security

Several of the libraries that we recommend to use for a real implementation are part of the standard library for that programming language, which is the case for both

java.security, *java.util*, and Golang's *net/http*. This means that these libraries are well tested and well used, which is positive in a security aspect since the packages are well supported and have undergone a lot of patching to make them more secure when vulnerabilities have occurred.

Multiple libraries are both owned and provided by large companies such as Google, Apple, and Microsoft. The combination of the financial support from big companies and the fact that these libraries are widely used gives an assurance regarding the security of these libraries since many resources are put into them in order to keep them secure. These companies also have a lot of experience when it comes to secure software, which gives more reasons for us to believe that these libraries are secure to use.

Multiple libraries are also open source and maintained by smaller companies and working groups. This is the case for *libfido2* provided by Yubico and *OpenSSL* provided by the OpenSSL community. These organisations are well regarded and trusted within the security community, which gives us reasons to believe that the libraries provided are secure to use. They are also libraries that are widely used in their respective environments.

7.4 Client Verification for User

Attestation can verify the client to the AS, but from the user's perspective, it does nothing to verify the client to the user. We wanted something that could fix that so the user would have some way to verify that the client is who it claims to be. The flow we designed in Figure 6.5 is optional to use and, if chosen, would only have to be used one time once the app is installed in order to verify the app to the user. The flow is not part of the standard protocol at all but only fills the niche of making the user sure that the client is legitimate. With it being optional and only needed once if used, there is no real adverse effect on the user experience.

7.5 User Experience

As mentioned in Section § 1.4, the analysis of the UX is done from our point of view and not from a real user's perspective.

The user experience depends on which of the three modes the user is currently using to authenticate themselves. When evaluating the user experience, we done not look directly at the security aspects of each flow since most users have no idea of how it actually works. Instead, we focus on only the parts that the users have to interact with.

For ROPC, users use their credentials and log in directly, which is currently already common, and users are already used to using it. While we still recommend against using this flow for security reasons, from a UX perspective, it is something users have used before.

CIBA requires users to leave the app temporarily. However, doing it in order to use, for example, BankID is already very common, and users do not have to learn anything new or do anything they have not done before. Also, if you are using BankID on the same device as they are trying to log in on, you can have smooth redirects to and from BankID. Since BankID in Sweden today is highly trusted both by developers and users, it provides a good user experience since the user trusts the login process.

FIDO2 could only require the user to put their thumb to the scanner on their phone. Everything else done is hidden from the user. Since users already like using their fingerprints to authenticate themselves, as mentioned in the study in Section § 3.4, this flow would provide the best user experience on its own. The fact that it also provides excellent security is just a bonus.

Overall, our protocol provides better UX compared to regular OAuth since there is no browser redirection, and in the case of the ROPC and FIDO2 flows, there is no redirection. The process of establishing the identity of the user can be integrated directly into the app, and in the case of FIDO2, it provides great security. When logging in with OAuth, the browser popping up leads to less than ideal UX since the user has no real idea what is popping up and has to trust that the login page displayed is trustworthy.

CIBA provides a similar UX to OAuth since there is still some redirect, either to something like BankID or the user having to open their mail. However, users would be more comfortable logging in to the highly trusted BankID compared to some arbitrary web page.

7.6 Downsides and Constraints

There are some modern hardware and software required on the mobile phone in order for the protocol to work, specifically for attestation. Android phones need hardware that became publicly available in 2017, and for iOS, the phone needs to be able to at least support iOS 14, which iPhone 6s and later models do. Since the features needed are required for the attestation, this means that if you have an old phone, then the protocol will not work. However, if the phone was released in 2017 or later, then the protocol would not fail due to hardware limitations. The security we can add with this restriction is worth the price of old phones being unable to use the protocol.

Since the DPoP RFC was only published in its final version in September 2023, there are only a few working implementations of it. This means that if you want to use it, you might have to build your own, and since it is new, there might be exploits that have not been discovered and patched yet. However, this is something that happens with every new protocol and will get better over time. The RFC has existed in older versions since 2020 before being officially published in 2023, so people have known

about it for some time.

In order for DPoP to work, the device the client runs on needs to be able to both generate and securely store cryptographic keys since if they get stolen or can be predicted, the whole protocol falls apart. Not all devices are capable of doing this, and as such, DPoP does not work on them. However, as previously stated, all modern phones can do this, which means that DPoP is possible to implement on phones today.

8 Conclusion

The purpose of this thesis was to design a secure login protocol for native applications on mobile phones that do not use browser redirects. This chapter answers the questions stated in Section § 1.2.

Is it possible to design an authentication/authorization protocol that follows the BCPs without browser redirection?

From our research, we can conclude that such a protocol can be designed. mAuth follows all the BCPs, which are not related to implementation details, thanks to the use of mechanisms such as attestation, DPoP with client instance keys, and keeping the essential parts of the code flow from regular OAuth. By extending the code flow in different parts of the flow and not modifying the base flow, we keep the essential security parts of that flow. Then, by using attestation and DPoP with instance keys, we can make sure that only a legitimate native app can talk with the AS and that the tokens issued by the AS can only be used by this legitimate client.

Since the protocol is only theoretical with accompanying pseudocode, it is hard to determine if a full implementation of such a protocol is possible. However, since both the theoretical protocol and the corresponding pseudocode show great promise, it provides a good indicator that a real implementation of this protocol would be possible and meet the security requirements placed upon it.

Can such a protocol also implement decoupled flows such as CIBA and FIDO2?

mAuth is designed to give developers the option to choose what type of flow they want to use depending on the security demands. This is fulfilled as step 11 in Figure 6.1 can be replaced by either of the integrated ROPC, CIBA, and FIDO2 flows.

How is user experience affected compared to regular OAuth?

The three different flows in mAuth provide different levels of user experience, but all three of them have similar or better UX compared to regular OAuth, in our opinion. The ROPC flow has the user log in with their credentials, like in OAuth, but without the browser redirection. CIBA with BankID has redirection but only to the trusted BankID app. In the case of the FIDO2 flow, the UX is a lot better than OAuth, only needing the user to provide some biometric credentials.

Summary

mAuth is a secure authorization/authentication protocol for mobile phones that provides both great security and UX with no browser redirects. mAuth also provides flexibility as a developer can choose between different flows to use depending on their security demands. The protocol is only theoretical with pseudocode at this point, but it shows promising potential.

8.1 Future Work

Future work would include doing an actual working implementation of our protocol, complete with an app, backend server, authorization server, and resource server. Then run tests and attacks against it to see if it holds up as well as we have claimed in this thesis. While we think it will hold up to common attacks, it is hard to tell if it actually will without a real implementation.

Another thing that could be done in a future project would be to study how the user experience would look with the implementation of our protocol. The analysis of the user experience was done from our point of view, but this does not paint the full picture since we have a better understanding of this protocol than regular users. It could be possible to perform a study using the theoretical version. However, we believe it would be hard to achieve any concrete result from this due to the protocol being theoretical and not an actual implementation that a user could use.

Another work that would also be interesting is to compare the user experience of our protocol with regular OAuth and see how these two experiences stand against each other. It would be interesting to see if we could improve on the current standard for authorization and authentication while still following the BCPs and other security recommendations.

Bibliography

[1] Android. *Play integrity API : google play : android developers*. Apr. 2023. URL: https://developer.android.com/google/play/integrity.

[2] Apple. *Apple iOS 14 devices*. 2024. URL: https://support.apple.com/sv-se/guide/iphone/iphe3fa5df43/14.0/ios/14.0.

[3] Apple. *Apple Secure Enclave*. 2024. URL: https://developer.apple.com/documentation/security/certificate_key_and_trust_services/keys/protecting_keys_with_the_secure_enclave.

[4] Apple. *DCAppAttestService*. 2024. URL: https://developer.apple.com/documentation/devicecheck/dcappattestservice.

[5] Apple. *Devicecheck*. Mar. 2024. URL: https://developer.apple.com/documentation/devicecheck.

[6] Apple. *Generating new cryptographic keys*. Apr. 2024. URL: https://developer.apple.com/documentation/security/certificate_key_and_trust_services/keys/generating_new_cryptographic_keys.

[7] Apple. *IOS 17*. Apr. 2024. URL: https://developer.apple.com/ios/.

[8] Auth0. *Authorization code flow with proof key for code exchange (PKCE)*. 2024. URL: https://auth0.com/docs/get-started/authentication-and-authorization-flow/authorization-code-flow-with-pkce.

[9] Auth0. *JSON web token claims*. Mar. 2024. URL: https://auth0.com/docs/secure/tokens/json-web-tokens/json-web-token-claims.

[10] Auth0. *JSON web token structure*. 2024. URL: https://auth0.com/docs/secure/tokens/json-web-tokens/json-web-token-structure.

[11] Auth0. *OAuth 2.0 Authorization Framework*. 2024. URL: https://auth0.com/docs/authenticate/protocols/oauth#grant-types.

[12] Auth0. *OpenID connect protocol*. 2024. URL: https://auth0.com/docs/authenticate/protocols/openid-connect-protocol.

[13] Dominick Baier, Joe DeCock, and Brock Allen. *Identitymodel/identitymodel .oidcclient*. Apr. 2024. URL: https://github.com/IdentityModel/IdentityModel.OidcClient?tab=readme-ov-file.

[14] John Bradley et al. "Web authentication:an API for accessing public key credentialslevel 3". In: *W3C* (Sept. 2023). Ed. by Michael B. Jones et al. URL: https://www.w3.org/TR/2023/WD-webauthn-3-20230927/#webauthn-client.

[15] Brian Campbell et al. "RFC 8705". In: *RFC 8705: OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens* (Feb. 2020). URL: https://www.rfc-editor.org/rfc/rfc8705.html.

[16] Curity. 2021. URL: https://curity.io/resources/documents/hypermedia-authentication-api-security-in-detail-whitepaper/.
[17] Curity. *Hypermedia authentication API*. 2023. URL: https://curity.io/resources/haapi/.
[18] Android Developers. *Android Keystore system : App quality : android developers*. Jan. 2024. URL: https://developer.android.com/privacy-and-security/keystore.
[19] Android Developers. *Download Android Studio and App Tools*. Apr. 2024. URL: https://developer.android.com/studio.
[20] Duende. *Duende Software Documentation*. Apr. 2024. URL: https://docs.duendesoftware.com/identityserver/v7/ui/ciba/.
[21] G Fernandez et al. In: *OpenID Connect Client-Initiated Backchannel Authentication Flow - Core 1.0* (Sept. 2021). URL: https://openid.net/specs/openid-client-initiated-backchannel-authentication-core-1_0.html.
[22] Daniel Fett. "FAPI 2.0 Attacker Model". In: *FAPI 2.0 Attacker model* (Dec. 2022). URL: https://openid.net/specs/fapi-2_0-attacker-model.html.
[23] Daniel Fett, Ralf Küsters, and Guido Schmitz. "A Comprehensive Formal Security Analysis of OAuth 2.0". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: Association for Computing Machinery, 2016, pp. 1204–1215. ISBN: 9781450341394. DOI: 10.1145/2976749.2978385. URL: https://doi.org/10.1145/2976749.2978385.
[24] Daniel Fett and Dave Tonge. "FAPI 2.0 Security Profile — draft". In: *FAPI 2.0 security profile - draft* (Nov. 2023). URL: https://openid.bitbucket.io/fapi/fapi-2_0-security-profile.html.
[25] Daniel Fett et al. "RFC 9449". In: *RFC 9449: OAuth 2.0 Demonstrating Proof of Possession (DPoP)* (Sept. 2023). URL: https://www.rfc-editor.org/rfc/rfc9449.html.
[26] Go. *HTTP*. Apr. 2024. URL: https://pkg.go.dev/net/http.
[27] Sunila Goray. *The history of mobile apps and evolution of Mobile Platforms*. Jan. 2024. URL: https://webandcrafts.com/blog/history-of-mobile-apps.
[28] E. Hammer-Lahav. *Introduction*. Sept. 2007. URL: https://oauth.net/about/introduction/.
[29] Dick Hardt. "RFC 6749: The oauth 2.0 authorization framework". In: *IETF Datatracker* (Oct. 2012). URL: https://datatracker.ietf.org/doc/html/rfc6749#section-1.7.
[30] Dick Hardt, Aaron Parecki, and Torsten Lodderstedt. "The Oauth 2.1 Authorization Framework". In: *IETF Datatracker* (Jan. 2024). URL: https://datatracker.ietf.org/doc/html/draft-ietf-oauth-v2-1-10.
[31] Torsten Lodderstedt, Mark McGloin, and Phil Hunt. "RFC 6819: Oauth 2.0 threat model and security considerations". In: *IETF Datatracker* (Jan. 2013). URL: https://datatracker.ietf.org/doc/html/rfc6819.

[32] Torsten Lodderstedt et al. "OAuth 2.0 security best current practice". In: *IETF Datatracker* (Oct. 2023). URL: https://datatracker.ietf.org/doc/html/draft-ietf-oauth-security-topics.

[33] Tobias Looker and Paul Bastian. "OAuth 2.0 Attestation-Based Client Authentication". In: *OAuth 2.0 attestation-based client authentication* (Feb. 2024). URL: https://vcstuff.github.io/draft-ietf-oauth-attestation-based-client-auth/draft-ietf-oauth-attestation-based-client-auth.html.

[34] Pedro Martelletto and Ludvig Michaelsson. *Yubico/libfido2: Provides library functionality for FIDO2, including communication with a device over USB or NFC.* Apr. 2024. URL: https://github.com/Yubico/libfido2.

[35] Microsoft. *Multi-Factor Authentication.* 2024. URL: https://support.microsoft.com/en-gb/topic/what-is-multifactor-authentication-e5e39437-121c-be60-d123-eda06bddf661.

[36] Microsoft. *System.IdentityModel.tokens.Jwt 7.5.1.* Apr. 2024. URL: https://www.nuget.org/packages/System.IdentityModel.Tokens.Jwt/.

[37] Okta. *What is OAuth 2.0 and what does it do for you?* 2024. URL: https://auth0.com/intro-to-iam/what-is-oauth-2.

[38] Inc. OpenSSL Foundation. Apr. 2024. URL: https://www.openssl.org/.

[39] Oracle. *Package java.security.* Jan. 2024. URL: https://docs.oracle.com/javase/8/docs/api/java/security/package-summary.html.

[40] Oracle. *Package java.util.* Jan. 2024. URL: https://docs.oracle.com/javase/8/docs/api/java/util/package-summary.html.

[41] Aaron Parecki, George Fletcher, and Pieter Kasselman. "OAuth 2.0 for First-Party Applications". In: *OAuth 2.0 for first-party applications* (Mar. 2024). URL: https://drafts.aaronpk.com/oauth-first-party-apps/draft-parecki-oauth-first-party-apps.html.

[42] PKCE. *Protecting Apps with PKCE.* Dec. 2021. URL: https://www.oauth.com/oauth2-servers/pkce/.

[43] PortSwigger. *BURP suite - application security testing software.* 2024. URL: https://portswigger.net/burp.

[44] PortSwigger. *Configuring an IOS device to work with BURP suite professional.* Jan. 2024. URL: https://portswigger.net/burp/documentation/desktop/mobile/config-ios-device.

[45] Martiño Rivera-Dourado et al. "A Novel Protocol Using Captive Portals for FIDO2 Network Authentication". In: *Applied Sciences* 14.9 (2024). ISSN: 2076-3417. DOI: 10.3390/app14093610. URL: https://www.mdpi.com/2076-3417/14/9/3610.

[46] Karsten Meyer zu Selhausen and Daniel Fett. "RFC 9207". In: *RFC 9207: OAuth 2.0 Authorization Server Issuer Identification* (Mar. 2022). URL: https://www.rfc-editor.org/rfc/rfc9207.html.

[47] Y. Sheffer, P. Saint-Andre, and T. Fossati. In: *Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)* (Nov. 2022). URL: https://www.rfc-editor.org/info/bcp195.

[48] Travis Spencer. *Travis Spencer - Software Engineer.* Jan. 2021. URL: https://travisspencer.com/articles/oauth-without-redirection/.

[49] Dave Tonge et al. "Financial-grade API: Client initiated Backchannel Authentication profile". In: *Draft-02: Financial-grade API: Client Initiated Backchannel Authentication Profile* (Aug. 2019). URL: https://openid.net/specs/openid-financial-api-ciba.html.

[50] Jeffery Walton et al. *Certificate and public key pinning*. Mar. 2024. URL: https://owasp.org/www-community/controls/Certificate_and_Public_Key_Pinning.

[51] Yubico. *FIDO2 passwordless authentication*. Dec. 2023. URL: https://www.yubico.com/authentication-standards/fido2/.

[52] Verena Zimmermann and Nina Gerber. "The password is dead, long live the password – A laboratory study on user perceptions of authentication schemes". In: *International Journal of Human-Computer Studies* 133 (2020), pp. 26–44. ISSN: 1071-5819. DOI: https://doi.org/10.1016/j.ijhcs.2019.08.006. URL: https://www.sciencedirect.com/science/article/pii/S1071581919301119.

A
Security Analysis

For the OAuth BCP described in section 2.5, the practices we **DO** follow are the following:

- **(2.1) Protecting Redirect Based Flows:** These recommendations are followed since the threats mentioned in this section requires the use of a browser, which is not the case in our protocol since we don't use a browser.

- **(2.1.2) Implicit Grant:** The implicit flow is not used in our protocol.

- **(2.2.1) Access Tokens:** This is satisfied through the use of DPoP and TLS.

- **(2.2.2) Refresh Tokens:** This is satisfied through the use of DPoP.

- **(2.3) Access Token Privilege Restriction:** By following the principle of least privilege, it satisfies this practice.

- **(2.5) Client Authentication:** This is satisfied through the use of DPoP and attestation.

- **(2.6) Other Recommendations:** This includes recommendations which are specific for implementation which is out of scope for this thesis. Other parts of this is satisfied through the use of TLS and the non-use of a browser.

- **(4.1.1) Redirect URI Validation Attacks on Authorization Code Grant:** This is satisfied as redirect URIs are not used.

- **(4.1.2) Redirect URI Validation Attacks on Implicit Grant:** Neither redirect URIs or the implicit grant is used.

- **(4.2.1) Leakage from OAuth Client:** Attackers can potentially steal authorization code or token, but can not use it because of DPoP and the fact that the authorization code is never sent to the client frontend.

- **(4.2.2) Leakage from the Authorization Server:** This is an implementation detail of the authorization server, which is out of scope for this thesis.

- **(4.3.1) Authorization Code In Browser History:** This is satisfied as a browser is not used and the authorization code is a one time use, which pro-

vides replay prevention.

- **(4.3.2) Access Token in Browser History:** This is satisfied as a browser is not used.
- **(4.4) Mix-Up Attacks:** This is satisfied by always checking the "iss" parameter for all requests sent and received from the AS. This makes sure that the client is always talks with the same issuer all the time. If there is a miss-match, then the flow will be aborted.
- **(4.5) Authorization Code Injection:** This is satisfied as the authorization code is only communicated between the authorization server and the client backend server, and not the frontend. This means that the authorization code is never exposed to the public client.
- **(4.6) Access Token Injection:** This is satisfied through the use of DPoP.
- **(4.7) CSRF:** CSRF requires the use of a browser for the attack to be possible. A browser is not used in our protocol, which means that this recommendation is met.
- **(4.9.1) Access Token Phising by Counterfeit Resource Server:** Our protocol protects from this by sender-constraining the access token through the use of DPoP.
- **(4.9.2) Compromised Resource Server:** Our protocol protects from this by sender-constraining the access token through the use of DPoP.
- **(4.10.1) Sender Constrained Access Tokens:** This is satisfied as our protocol uses DPoP.
- **(4.10.2) Audience-Restricted Access Tokens:** This is satisfied by following the principle of least privilege when generating access tokens.
- **(4.11.1) Client as Open Redirector:** Our protocol do not use open redirectors.
- **(4.11.2) Authorization Server as Open Redirector:** Our protocol do not use open redirectors.
- **(4.12) 307 Redirect:** This is an implementation detail which is out of scope for this thesis.
- **(4.13) TLS Terminating Reverse Proxies:** This is an implementation detail which is out of scope for this thesis.

- **(4.14) Refresh Token Protection:** This is satisfied through the use of TLS and DPoP.

- **(4.15) Client Impersonating Resource Owner:** This is an implementation detail which is out of scope for this thesis.

- **(4.16) Clickjacking:** This is not a problem for our protocol since a browser is not utilized.

- **(4.17) Authorization Server Redirecting to Phising Site:** This is not an issue as our protocol do not use redirection.

- **(4.18.1.1) Insufficient Limitation of Receiver Origins:** This is not a problem since our protocol do not use a browser.

- **(4.18.1.2) Insufficient URI Validation:** This is not a problem since our protocol do not use a browser.

- **(4.18.1.3) Injection after Insufficient Validation of Sender Origin:** This is not a problem since our protocol do not use a browser.

For the OAuth BCP described in section 2.5, the practices we **DO NOT** follow are the following:

- **(2.1.1) Authorization Code Grant:** This is not followed since PKCE is not used in our protocol.

- **(2.4) ROPC Grant:** This practice is not followed since we incorporate the ROPC flow in our protocol. However, we recommend to not use it.

- **(4.8) PKCE Downgrade Attack:** This is not satisfied as we don't support PKCE.

For the OAuth Threat Model described in section 2.7, the practices we **DO** follow are the following:

- **(4.1.1) Obtaining Client Secrets:** Our protocol does not use client secrets, therefore there is no secret to steal.

- **(4.1.2) Obtaining Refresh Tokens:** Potential refresh tokens are DPoP bound and cannot be used even if stolen. Even so tokens should be held in secure storage. If device is cloned private key used to generate DPoPP is not cloned with the device.

- **(4.1.3) Obtaining Access Token:** Access tokens are DPoP bound and cannot be used even if stolen. Store the tokens as securely as possible, and limit their scope.

- **(4.1.4) End-User Credentials Phished Using Compromised or Embedded Browser:** If using the CIBA or FIDO2 flows then credentials are never entered into the app and thus cannot be stolen.
- **(4.1.5) Open Redirectors on Client:** Our protocol does not use redirects.
- **(4.2.1) Password Phising by Counterfeit AS:** Our protocol always uses TLS and will not accept connections from regular HTTP.
- **(4.2.2) User Unintentionally Grants Too Much Access Scope:** Explain scope clearly to user.
- **(4.2.3): Malicious Client Obtains Existing Authorization by Fraud:** Always check the DPoPP and never automatically say yes to requests.
- **(4.2.4) Open Redirector:** Our protocol does not use redirects.
- **(4.3.1) Eavesdropping Access Tokens:** Our protocol always uses TLS. Even in stolen access tokens are DPoP bound.
- **(4.3.2) Obtaining Access Tokens from AS Database:** Make sure to secure your database. Also even in stolen access tokens are DPoP bound.
- **(4.3.3) Disclosure of Client Credentials during Transmission:** Our protocol always uses TLS.
- **(4.3.4) Obtaining Client Secret from Authorization Database:** Our protocol does not use client secrets.
- **(4.3.5) Obtaining Client Secret by Online guessing:** Our protocol uses strong client authentication by attestation and does not use client secrets.
- **(4.4.1.1) Eavesdropping or Leaking Authorization codes:** Our protocol always enforces TLS. The code is only sent from AS to the client backend server meaning it cannot be stolen by an attacker intercepting traffic to and from the client frontend. It is also only usable for a short while.
- **(4.4.1.2) Obtaining Authorization codes from AS database:** Our auth codes do not need to be stored in order to later be verified.
- **(4.4.1.3) Online Guessing of Authorization codes:** High entropy codes are hard to guess. The codes have a short lifespan so has to be guessed in between being issued and expiring or used. Will also only lead to getting an access token the is DPoP bound has thus cannot be used by the attacker.
- **(4.4.1.4) Malicious Client Obtains Authorization:** Our protocol au-

thenticates the client before doing anything else meaning there is no way for a malicious client to pretend to be a legitimate client in front of the AS.

- **(4.4.1.5) Authorization code Phishing:** The auth code is only sent to the backend server and thus cannot be intercepted in front of the frontend. All traffic is also always protected by TLS.
- **(4.4.1.6) User Session Impersonation:** Only relevant for web session which our protocol does not use.
- **(4.4.1.7) Authorization code leakage through Counterfeit Clients:** With the attestation step in our protocol the risk of counterfeit clients successfully impersonation legitimate ones is low.
- **(4.4.1.8) CSRF Attack against redirect-uri:** Our protocol does not use redirect uri.
- **(4.4.1.9) Clickjacking Attack against Authorization:** Only a threat against browsers and thus not relevant for us.
- **(4.4.1.10) Resource owner Impersonation:** The attestation process protects the AS from malicious clients.
- **(4.4.1.11) DoS Attacks That Exhaust Resources:** Limit that amount of codes and tokens can be granted to each user. High entropy in code generation.
- **(4.4.1.12) DoS Using Manufactured Authorization codes:** Rate-limit clients that fail too much. The client frontend also gets no auth codes.
- **(4.4.1.13) Code Substitution (OAuth Login):** Client gets authenticated so a malicious app cannot log in even with correct credentials. The frontend client also get no auth code.
- **(4.4.2) Implicit Grant:** This flow is not used by our protocol at all so all threats targeting it are not relevant for us and as such have been omitted from the security analysis.
- **(4.4.3) Resource Owner Password Credentials (ROPC):** We recommend against using this flow but if it has to be used only use it for first party clients with TLS. Tell users not to have the same password at other organizations.
- **(4.4.3.1) Accidental Exposure of Passwords at Client Site:** When sending the credentials hash them first. If there is risk of them showing up in a log then hash them before logging them as well.

- **(4.4.3.2) Client Obtains Scopes without End-User Authorization:** Clients always get authenticated first meaning no intentional abuse of scope.
- **(4.4.3.3) Client Obtains Refresh Token without End-User Authorization:** We authenticate the client and it only gets reference tokens not the real tokens. Also we do not use automatic authentication.
- **(4.4.3.4) Obtaining User Passwords on Transport:** Always use TLS. Do not send passwords in plaintext.
- **(4.4.3.5) Obtaining User Passwords from Authorization Server Database:** Use best current practices when designing your database. Out of scope for this protocol.
- **(4.4.3.6) Online Guessing:** Try to force users to use secure passwords or passphrases. Lock accounts that try too many times by using a "tar pit" to increase lockout times with every subsequent attempt.
- **(4.4.4) Client Credentials:** Use the same countermeasures as in 4.4.3.
- **(4.5.1) Eavesdropping Refresh Tokens from AS:** Always use TLS. Any request send not under TLS gets rejected.
- **(4.5.2) Obtaining Refresh Token from AS Database:** Use best current practices when designing your database. Out of scope for this protocol.
- **(4.5.3) Obtaining Refresh Token by Online Guessing:** Generate tokens with high entropy that are also always DPoP bound and thus cannot be used even if guessed.
- **(4.5.4) Refresh Token Phishing by Counterfeit AS:** Always use TLS.
- **(4.6.1) Eavesdropping Access Tokens on Transport:** Always use TLS. Tokens are DPoP bound and thus cannot be used even if stolen.
- **(4.6.2) Replay Authorized Resource Server Request:** The request has to have accompanying DPoP for it to be valid. Always use TLS to make it harder to know what is being replayed. Also always check if the same request has been seen before.
- **(4.6.3) Guessing Access Tokens:** Generate tokens with high entropy to make harder to guess. The tokens are also DPoP bound and cannot be used by attacker if stolen. Keep the lifetime of the token short.
- **(4.6.4) Access Token Phishing by Counterfeit RS:** Tokens are DPoP bound and cannot be used by attacker even if stolen. Also authenticate the

RS but that is out of scope for this protocol.

- **(4.6.5) Abuse of Token by Legitimate RS or Client:** Restrict tokens to only certain resource servers. For example a token that can be used to access the photo RS should not be able to be used at the video RS.

- **(4.6.6) Leak of Confidential Data in HTTP Proxies:** This is an implementation detail, which is out of scope for this thesis.

- **(4.6.7) Token Leakage via Log Files and HTTP Referrers:** Tokens are DPoP bound and cannot be used by attacker even if stolen.

For the OAuth Threat Model described in section 2.7, the practices we **DO NOT** follow are the following:

- **(4.1.4) End-User Credentials Phished Using Compromised or Embedded Browser:** If using the ROPC flow then credentials can be stolen from a malicious version of the app. We recommend against using this flow but do not forbid it entirely.

For the FAPI Security profile, described in section 2.6, the practices we **DO** follow are the following:

- **(5.3.2) Requirements for authorization servers:** This partly followed by only allowing the client backend to talk to the AS, by using DPoP, server provided nonce and only accepting an authorization code once.

- **(5.3.3) Requirements for clients:** This is partly followed through the use of DPoP and the fact that a browser is not used, which eliminates the those related risks.

- **(5.3.4) Requirements for resource servers:** This practice is followed with the use of DPoP, but most requirements described are implementations details which is out of scope for this thesis.

- **(5.4) Cryptography and secrets:** This practice is followed, but the details are implementation specific which is out of scope for this thesis.

- **(5.5) MTLS protection of all endpoints:** This is not relevant to our protocol as it is also stated to be out of scope for the FAPI standards.

- **(6.1) Access token lifetimes:** This is followed by setting the lifetime of access token as low as possible depending on the situation and also allowing rotation of refresh tokens in order to refresh the sender-constraining key.

- **(6.2) DPoP proof replay:** This is not relevant as this requires an attacker of type A5, which is not relevant for FAPI 2.0 as stated in the FAPI Attackermodel [22].

- **(6.3) JWKS URIs:** This is an implementation detail which is out of scope for this thesis.

- **(6.4) Duplicate key identifiers:** This is an implementation detail which is out of scope for this thesis.

- **(6.5) Injection of stolen access tokens:** This is followed through the use of DPoP and the fact that a new key pair used in the DPoPP is generated for each login session, thus making it unique for each AS.

- **(6.6) Authorization requests leaks lead to CSRF:** This is not a relevant problem for our protocol since it does not utilize a browser, which is a requirement for a CSRF attack to occur.

- **(6.7) Browser-swapping attacks:** The protocol protects against this as no browser is used.

- **(6.8) Incomplete or incorrect implementations of the specifications:** This is an implementation detail which is out of scope for this thesis.

For the FAPI Security profile, described in section 2.6, the practices we **DO NOT** follow are the following:

- **(5.3.2) Requirements for authorization servers:** This is partly not followed since it specifies that the ROPC must be rejected, which we don't do. We recommend to not use it but do not reject it entirely. PKCE is also not used in our protocol.

- **(5.3.3) Requirements for clients:** This is partly not followed since our protocol do not utilize PKCE.

For the FAPI CIBA profile, described in section 2.6.5, the practices we **DO** follow are the following:

- **(5.2.2) Authorization Server:** Many of these requirements are implementation details which we do not consider, but we follow those that are relevant. The client backend acts as a confidential client and the authorization server only has to support it. Also, only poll mode is supported, and maybe ping if necessary.

- **(5.2.3.1) Confidential Client General Provisions:** This is an implementation detail, which is out of scope for this thesis.

- **(6.2) Client Provisions:** This is an implementation detial which is out of scope for this thesis.

- **(7.2) Authentication sessions started without a users knowledge or**

consent: This is satisfied as we make sure to if possible, have the authentication device be the same device as the client is running on. If another device is used, then a QR code scan is used. If that is not possible, then a login hint such as personal number will be used. This approach mitigates this attack.

- **(7.3) Reliance on user to confirm binding messages:** If a genuine party and a malicious party send an authentication request at the same time to the same device, then both requests will be blocked and a timeout will occur such that the authentication device will not accept authentication requests for some amount of time.

- **(7.4) Loss of fraud markers to OpenID provider:** This is not a relevant problem as our protocol does not use any redirect based-flows.

- **(7.5) Incomplete or incorrect implementations of the specifications:** This is an implementation detail which is out of scope for this thesis.

- **(7.6) JWS/JWE Algorithm considerations:** This is an implementation detail which is out of scope for this thesis.

- **(7.7) Authentication Device security:** This is an implementation detail which is out of scope for this thesis.

- **(7.8) CIBA token delivery modes:** This is satisfied as we recommend to if possible, always use the poll mode. If that is not possible, ping mode may be used.

- **(7.9) TLS Considerations:** This is an implementation detail which is out of scope for this thesis.

- **(7.10) Algorithm considerations:** This is an implementation detail which is out of scope for this thesis.

- **(7.11) Encryption algorithm considerations:** This is an implementation detail which is out of scope for this thesis.

B

Pseudocode

B.1 Client Frontend

```
# Client Frontend

# Get challenge
challenge = get_challenge(client_id, backend_address)

# Generate public/private key pair
private_key, public_key = generateKeyPair()

# Store private key in secure hardware
store_key(private_key)

# Play Integrity API or AppAttest
attestation_token = get_attestation(challenge, public_key)

# Generate DPoP proof
DPoPP = generate_DPoPP()

# Authorization code request
authorization_code_request(attestation_token, DPoPP,
    backend_address)

## ROPC ##
# Prompt user for credentials
if (received_ropc_login):
    credentials = get_user_credentials()
    DPoPP = generate_DPoPP()
    post_credentials(credentials, DPoPP, backend_address)

## CIBA ##
# Authentication request
if (received_CIBA_login):
    post_authentication_request(backend_address)

if (received_ACK):
    # CIBA flow has started. Wait for reference tokens or some kind
     of error.

# If email OTP is used
if (received_request_email_code):
    email_code = get_email_code()
    send_email_code(email_code, backend_address)
```

```
40
41 ## FIDO2 ##
42 # Sign challenge
43 if (received_challenge):
44     start_authenticator(challenge)
45     # Wait for user to authenticate
46     send_signed_challenge(backend_address)
47
48 ## Token Request ##
49 # Make token request
50 if (received_reference_token):
51     DPoPP = generate_DPoPP()
52     resource_request(reference_token, DPoPP, scope, backend_address
   )
53
54 if (received_resource):
55     use_resource()
56     # The flow is now done
```

B.2 Client Backend

```
1 # Client Backend
2
3 # Challenge request
4 if (received_challenge_request):
5     forward_challenge_request(AS_address)
6
7 # Challenge response
8 if (received_challenge_response):
9     forward_challenge_response(challenge, client_address)
10
11 # Authorization code request
12 if (received_code_request):
13     forward_code_request(attestation_token, DPoPP, AS_address)
14
15 ## ROPC ##
16 # Request credentials
17 if (received_ropc_login):
18     start_ropc_login(client_address)
19
20 # Post credentials to AS
21 if (received_ropc_credentials):
22     forward_ropc_credentials(credentials, DPoPP, AS_address)
23
24 ## CIBA ##
25 # Start CIBA
26 if (received_CIBA_login):
27     forward_CIBA_login(client_address)
28
29 # Autentication request
30 if (received_authentication_request):
31     forward_authetication_request(AS_address)
32
33 # Ack
34 if (received_ACK):
```

```
35      forward_ACK(client_address)
36
37      # Email OTP
38      if (receieved_request_email_code):
39          forward_request_email_code(client_address)
40
41      # First poll message
42      send_poll_message(AS_address)
43
44      while (True):
45          # Email OTP
46          if (receieved_email_code):
47              forward_email_code(email_code, AS_address)
48
49          # Polling sequence
50          receive_poll_response()
51
52          if (final_poll_response):
53              break
54          send_poll_message(AS_address)
55
56  ## FIDO2 ##
57  # Challenge response
58  if (received_challenge):
59      forward_challenge(challenge, client_address)
60
61  # Signed challenge
62  if (received_signed_challenge):
63      forward_signed_challenge(signed_challenge, AS_address)
64
65  ## Token Request ##
66  # Make token request
67  if (received_authorization_code):
68      make_token_request(auth_code, AS_address)
69
70  # Store access token, send reference token
71  if (received_token):
72      store_token(access_token)
73      send_reference_token(reference_token, client_address)
74
75  # Make resource request
76  if (received_resource_request):
77      get_access_token(reference_token)
78      forward_resource_request(access_token, DPoPP, scope,
        API_address)
79
80  # Resource response
81  if (received_resource):
82      forward_resource(resource, client_address)
```

B.3 Authorization Server

```
1  # Authorization Server
2
3  # Challenge request
```

```
4  if (received_challenge_request):
5
6      # Challenge is a high entropy nonce
7      challenge = generate_challenge()
8      send_challenge(challenge, backend_address)
9
10 # Authorization code request
11 if (received_code_request):
12     verify(attestation_token)
13     verify(DPoPP)
14
15     ## ROPC ##
16     if (ROPC):
17         # Initiate ROPC
18         start_ropc_login(backend_address)
19
20         # Verify user credentials
21         if (received_ropc_credentials):
22             verify(DPoPP)
23             verify(credentials)
24             if (legit_credentials):
25                 send_auth_code(auth_code, backend_address)
26
27     ## CIBA ##
28     if (CIBA):
29         # Initiate CIBA
30         start_CIBA_login(backend_address)
31
32         # Begin authentication process
33         if (received_authentication_request):
34             send_ACK(backend_address)
35             match mode:
36                 case bankID:
37                     start_BankID()
38                     wait_for_BankID()
39                 case email_link:
40                     send_email_link()
41                     wait_for_link()
42                 case email_code:
43                     send_email_code()
44                     request_email_code(backend_address)
45                     if(received_email_code):
46                         verify(email_code)
47
48         # While waiting for bankID/email, polling happens
49         if (received_poll_message):
50             # If user has completed authentication
51             if (authentication_done):
52                 send_auth_code(auth_code, backend_address)
53             # Keep polling
54             else:
55                 send_poll_message(backend_address)
56
57     ## FIDO2 ##
58     if (FIDO2):
59         # Challenge
```

```
        challenge = generate_challenge()
        send_challenge(challenge, backend_address)

        # Verify signature
        if (received_signed_challenge):
            verify(signed_challenge)
            if (challenge_valid):
                send_auth_code(auth_code, backend_address)

## Token Request ##
# Verify token request
if (received_token_request):
    verify(auth_code)
    if (auth_code_valid):
        send_token(access_token, backend_address)

# Verify resource request from API
if (received_verification_request):
    verify(DPoPP)
    verify(access_token)
    send_verification_result(verification_result, API_address)
```

B.4 API

```
# API

# Verify the resource request
if (received_resource_request):
    send_verification_request(DPoPP, access_token, AS_address)

# Send the resource to the requesting party
if (received_verification_result and result_valid):
    verify_scope()
    resource = get_resource()
    send_resource(resource, backend_address)
```