

Login hardening with Multi-factor Authentication

MICHAELA BERGMAN

MASTER'S THESIS

DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY

FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY



Login hardening with Multi-factor Authentication

Michaela Bergman
elt12mlu@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisor: Ben Smeets

Supervisor at Axis Communications: Reza Shams Amiri

Examiner: Tomas Johansson

June 23, 2021

Abstract

The aim with this Master's Thesis work was to conduct research about different available authenticators, as well as implementing a multi-factor authenticator into the currently used login-application. The research included biometric authentication technologies, and investigation of how to implement these to create a highly secure and customizable multi-factor authentication for My Axis-accounts for Axis Communications in Lund. A part of the research was to investigate and compare weaknesses, vulnerabilities, trade-offs, practical considerations, and security storage for common authenticators, as well as recovery and support for the loss of an authentication factor. The method used to achieve these goals was to collect information from research papers, books, and studies about authentication and authenticators, to do a small study about account recovery, and to investigate and analyze the currently used authentication system to implement a multi-factor authenticator that extends the system. The key objectives of the project were gaining an in-depth knowledge of multi-factor authentication, the OpenID Connect protocol and how it can be used in a system, and the implementation of a multi-factor authenticator that would utilize a combination of username/password authentication (knowledge factor), a smartphone (ownership factor), and biometric authentication (biometric factor). The implementation consists of a plugin that extends the current system and an Android application. The application authenticates the user with the built-in fingerprint sensor and creates a time-based one time password (TOTP), i.e., the application is an authenticator that combines TOTP with fingerprint so that the fingerprint never leaves the device, which mitigates the risk of biometric factor leaks. A key conclusion of this project is that the security level of the authenticator is decreased to the security level of its fallback method, in the case where the fallback method is less secure. This fallback method is used in case the user loses, for example, its email address or device.

Preface

This project was carried out during the winter and spring of 2020-2021 at Axis Communications in Lund. Currently Axis has an authentication solution in use requiring a traditional username and password login to access their personal My Axis-accounts. The basis for this research and implementation project was to increase the account security for My Axis-accounts, by implementing a multi-factor authenticator into the Axis external accounts, which will provide the user to choose between different authenticators to prove their identity when signing in through various applications.

First and foremost, I want to thank my supervisor Reza Shams Amiri at Axis. You have been a mentor, a teacher, and a friend during these weeks at Axis. A big thank you to the User Services team, for all the support and help, all the fikas and team activities, all the daily scrums, and above all, for making me feel like a part of your team during these isolated Corona-times.

I also want to say thank you to my supervisor at LTH, Ben Smeets. Your course in Advanced Computer Security got me interested in computer security and multi-factor authentication, and your knowledge about multi-factor authentication has been very valuable during this project.

Contents

1	Introduction	1
1.1	Methodology	2
2	Multi-factor Authentication	5
2.1	Commonly used Authenticator Methods based on Authentication Factors	6
3	The OAuth 2.0 Authorization Framework, RFC6749	21
3.1	Client Types, Profiles, Registration, Authentication and Identifier	23
3.2	Protocol Endpoints	24
3.3	Tokens	24
3.4	HTTP Redirections	26
3.5	Authorization Code Flow	27
3.6	Implicit Flow	28
3.7	Resource Owner Password Credentials Flow	29
3.8	Client Credentials Flow	30
4	OpenID Connect Protocol	33
4.1	Terminology	33
4.2	Authentication	34
4.3	Requesting Claims using Scope Values	34
4.4	Client Authentication	35
4.5	ID Token	37
4.6	UserInfo Endpoint	39
4.7	Authorization Code Flow	41
4.8	Implicit Flow	42
4.9	Hybrid Flow	43
5	Proof Key for Code Exchange by OAuth Public Clients	45
5.1	Authorization Code Interception Attack Flow	45
5.2	PKCE Protocol Flow	46
6	Security Threats	51
6.1	Cross-Site Request Forgery	51
6.2	Brute Force Attacks	53

6.3	Replay Attack	53
6.4	Clickjacking	54
7	Results _____	55
7.1	The Authentication and Authorization Process	55
7.2	Multi-Factor Authenticator	58
7.3	Implementation	62
8	Conclusion _____	69
8.1	Improvements and Changes	70
	Bibliography _____	73
A	The OAuth 2.0 Authorization Framework _____	79
A.1	Authorization Code Flow	79
A.2	Implicit Flow	82
A.3	Resource Owner Password Credentials Flow	85
A.4	Client Credentials Flow	86
B	OpenID Connect 1.0 _____	89
B.1	Authorization Code Flow	89
B.2	Implicit Flow	97
B.3	Hybrid Flow	100

List of Figures

3.1	An overview of the authorization code flow.	27
3.2	An overview of the implicit flow.	28
3.3	An overview of the resource owner password credentials flow.	30
3.4	An overview of the client credentials flow.	30
4.1	An overview of Authentication using the Authorization Code Flow.	41
4.2	An overview of Authentication using the Implicit Flow.	43
4.3	An overview of Authentication using the Hybrid Flow.	44
5.1	An overview of an Authorization Code Interception Attack.	45
5.2	An overview of the PKCE Protocol Flow.	47
6.1	An overview of a Cross-Site Request Forgery Attack.	51
7.1	An overview of the authorization code flow and where the authentication by the multi-factor authenticator will be executed.	57
7.2	The page of the HTML-login.	63
7.3	The page that is displayed to the user if there is no device registered for the user.	64
7.4	The registration page.	64
7.5	The authentication page.	65
7.6	App view, Authentication Failed	66
7.7	App view, Authentication Help	67
7.8	App view, Authentication Success	68

List of Tables

2.1	Account recovery strategies of 10 popular web services.	16
2.2	Authentication time for common authenticator methods [33].	18
4.1	OpenID Connect Authentication Flows	34

Introduction

The CIA triad is a model that describes the three key objectives or goals of information security: Confidentiality, Integrity and Availability. Confidentiality limits access to information, i.e., only those with permission to a resource are able to get access to it. Integrity assures that the data is trustworthy and accurate, and that the data has not been modified by unauthorized users. Availability ensures that authorized users have reliable access to data when they request it. Examples of computer attacks effecting CIA are malicious software, phishing, and denial of service. The three core services in an ICT (Information and Communications Technology) system that support the methods to distinguish between authorized and non-authorized user coupled to the CIA model are referred to as AAA: Authentication, Authorization, and Accountability. While there are other services that have a role in achieving the CIA objectives we will in this thesis focus on authentication.

Authentication is the process of uniquely identifying an individual, authorization is the process of specifying access rights to resources (data, information, services, files, networks, etc.). Accountability enables a system administrator to track and monitor system interactions of individuals. [20]

User authentication (sometimes also just called identification) is a process that consists of two parts: identity claiming and verification [39]. Identity claiming refers to the process where the user, sometimes called the claimant, presents an identifier to a security system, e.g. a username. Verification means that the user verifies its claimed identity, e.g., a password. There are different means of authentication (authentication factors), which can be used successively and this is referred to as multi-factor authentication. [21] Systems that rely on single factor authentication have been found to be vulnerable to different security attacks, and in order to increase the security there has been an increase in usage of multi-factor authentication [38]. The selection of authentication factors that are used in an authentication process determines the performance of the multi-factor authenticator as a whole [29].

Today, Axis uses a traditional username and password login procedure for users to access their personal My Axis-accounts. The My Axis-accounts are available for anyone that has a unique email address, and during registration an email verification is needed. If the account holder's password is compromised, the user will

no longer be able to authenticate itself. Therefore, to increase account security, Axis would like to implement multi-factor authentication into the Axis external accounts, which will provide the user to choose between different authenticators to prove their identity when signing in through various applications, e.g., fingerprint authentication or a one-time password.

The goals of this project are:

- Gain in-depth knowledge of the OpenID Connect Protocol and Multi-Factor Authentication, since these protocols will be central in our approach to integrate a flexible and secure authentication framework.
- Find a highly secure and user-friendly authentication method for a certain use case. Therefore, conduct research about common authentication methods and technologies based on authentication factors: **Knowledge factors** (passwords, PINs, security questions), **Possession factors** (ID cards, security tokens, one-time passwords, smartphones), **Biometric factors** (fingerprint, retina, iris, hand geometry, facial characteristics). In particular look at the following aspects:
 - Weaknesses, vulnerabilities, trade-offs, practical considerations and security token storage
 - Recovery and support for lost authentication factors
- Implement a multi-factor authenticator into the currently used login-application

This report is organized as follows: A brief methodology section, see Section 1.1, is included in this introduction chapter with information about the theories, methods, and tools that were applied in the project. Chapter 2 gives an introduction to multi-factor authenticators based on different authentication factors, a shorter study of the account recovery process of 10 common web services, and a comparison of commonly used authenticators. Chapters 3, 4 and 5 give a summary about the OAuth 2.0 protocol, the OpenID Connect protocol, and the Proof Key for Code Exchange by OAuth Public Clients protocol, respectively. These protocols are widely used and describe the processes of authentication and authorization. Some security threats that are related to the authentication and authorization process are described in Chapter 6. The results and the conclusions of this project are given in Chapters 7 and 8, respectively.

User (entity) authentication can be divided into two types depending on the type of user: machine by machine, or machine by human (user authentication) [29]. The focus of this project is on user authentication. Particularly, the scope of the project is limited to user authentication in the context of a login process, i.e., login to a web service.

1.1 Methodology

We start our research on different available authenticators and technologies, to investigate how to implement a highly secure multi-level authentication, by acquiring

in-depth knowledge of available authentication methods and technologies. To limit the scope of the project, the general concepts and technologies of the authenticators were studied, rather than the authenticators that are available on the market. To gain knowledge, both primary and secondary data was collected. Primary data means collecting data from the founders of the authenticators or technologies, and peer-reviewed articles and papers. Secondary data means collecting data from other resources, e.g., applications that are using a specific authenticator method or technology.

In parallel with the research, a prototype of a multi-factor authentication was implemented into the currently used login-application. The implementation was an evaluation and investigation of the data collected. As a consequence we need to acquire an in-depth understanding of the OAuth 2.0 and OpenID protocols. To implement the multi-factor authenticator that extends the currently used authentication system, the system was investigated and analyzed. A Linux based laptop with all the necessary apps needed for development was used, as well as an Android device, Samsung Galaxy S8.

Finally, knowledge from research and experience from the implementation work was used to arrive at a set of conclusions and recommendations for further improvements.

Multi-factor Authentication

This chapter contains an introduction to multi-factor authentication. Specifically we present commonly used authenticators (authentication methods or technologies) based on authentication factors, a shorter study of the account recovery process of 10 common web services, and a comparison between the commonly used authenticators.

Authentication contains the process of verifying the identity of someone or something, in our case a user. Organizations can, by the process of authentication, keep their networks more secure by permitting only authenticated users to access protected resources.

The user authentication process consists of an identity claim and a verification process of a user. The identity claim consists of a user presenting an identifier to a security system, e.g., a username. Verification means that the user verifies its claimed identity, e.g., a password. There are different means of authentication (authentication factors), which can be used successively and this is referred to as multi-factor authentication. [21] Due to a number of security threats, it was realized that authentication with just a single factor is not reliable to provide adequate protection [27]. Therefore, two-factor authentication (2FA) was proposed, which couples the representative data (username/password combination) with the ownership factor, such as a smartcard or a phone [28], [30]. Systems that rely on single factor authentication have been found to be vulnerable to different security attacks, and in order to increase the security there has been an increase in usage of multi-factor authentication [38]. The selection of which authentication factors are used in an authentication process determines the performance of the multi-factor authenticator as a whole [29].

One main challenge with MFA is the correlation between the user instance and the instance of the smart sensor on the device (e.g., fingerprint sensor or face recognition camera on a smartphone). The correlation between the user instance and the instance of the application or sensor in the smartphone must be established for security reasons. It is important that only the authenticated user, which is authenticated with the first-factor authenticator, has access rights to the application or sensor. There are also trade-offs to be made, mainly between usability and security, that need to be considered when integrating and/or implementing a

multi-factor authenticator. [22]

2.1 Commonly used Authenticator Methods based on Authentication Factors

Different authentication factor groups that are available today to verify the identity of a user is:

- **Something that the user knows** : Password, passphrase, PIN
- **Something that the user has**: Smart card, physical key, smartphone, personal ID card
- **Something that the user is**: fingerprint, retina, iris, hand geometry, facial characteristics
- **Something that the user does**: voice recognition, handwriting, behaviour pattern, typing rhythm
- **Where the user is**: geolocation security checks

The commonly used authenticator methods that will be investigated and compared in this chapter are based on one of the following authentication factor groups: Knowledge Factor (what the user knows), Ownership Factor (what the user has), and Biometric Factor (what the user is).

2.1.1 Commonly used Authenticator Methods based on Knowledge Factors

One commonly used authentication method that is based on knowledge factors is the traditional username/password authenticator. This authentication method was initially, and still is, used because of its simplicity and user friendliness [22]. Authentication methods that are based on knowledge factors usually work in the same way as the username/password authenticator, i.e., a HTML login form, except that the password is replaced with, for example, a PIN.

A large-scale study of web password habits [23], published in 2007, have found that the average user:

- logs into 25 different online services in the course of a three month period,
- only has seven passwords,
- and reuses one password for about three accounts on average.

This means that if one password is compromised, usually more than one protected resource will be unprotected. If the password is shared, it is no longer a safe authentication method since the account can be compromised immediately. Even if a password is not shared, an unauthorized user can gain access to the account by utilizing the dictionary attack, rainbow tables, or social engineering techniques [22]. The complexity (and length) of the password should have a minimum requirement

enforced by the system if a username/password authenticator is used [31].

The password has to be stored and connected to the user ID. Here, two approaches are used depending on how the authentication verification process is realized. One approach of verifying the knowledge of the password is to request the user to send the password (assuming that sending/entering the password does not lead the password is leaked). For this type of approach, the password should never be stored in plain text. The best practices of storing a password includes generating a random string *salt* that is concatenated with the password to protect against dictionary attacks. This generated string should then be hashed with a cryptographic hash function (e.g., SHA256), and the hashed string is then stored instead of the plain text password. [32] The other type of approach is that the security systems stores the password and conducts a so called challenge-response scheme. Here the user is requested to send a response that is computed from the password and challenge value it received. In this case, the security system must be designed to hold the passwords securely confidentiality protected.

2.1.2 Commonly used Authenticator Methods based on Ownership Factors

A study on five common two-factor authentication methods [33] showed that common authentication methods or technologies based on ownership factors are:

- SMS
- Time-based One Time Password (TOTP)
- Pre-generated codes
- Push notifications
- U2F Security Keys

However, all the above methods are two-factor authenticators, and not one-factor authenticators, since they both include something that the user has (a mobile phone or a security key) and something that the user knows (a verification code or a TOTP). The following sections is a summary of the presented two-factor authentication methods in the study.

SMS

The user is sent a one-time verification code through a text message to their mobile phone. The reason why this authenticator is common is because most users already have a mobile phone that can receive text messages.

Usability advantages and disadvantages:

- Advantages:
 - most users already has a mobile phone that can receive text message (99% of US)

- Disadvantages:
 - delayed delivery
 - lack of cellular service
 - miscopying the code from phone to computer

Security threats:

- man-in-the-middle attack (SIM-swapping attack), mobile networks do not encrypt SMS messages in transit
- brute force attack on a stolen hashed code, since the server (relying party) must store the one-time code (salted and hashed) for later verification while the SMS message is sent
- targeted phishing attacks of SMS codes

TOTP

A time-based one time password (TOTP) is usually generated through a smartphone application. By scanning a QR code, the smartphone gets the secret key from the provider. A TOTP is generated by hashing and truncating the value of a combination of the secret with a time-interval. The result is typically a 6 or 7 digits long human readable code. The server verifies the TOTP by using the same method that is used to generate the TOTP.

Usability advantages and disadvantages:

- Advantages:
 - no delayed deliver
 - works with any kind of internet connectivity
 - not requiring cellular service
- Disadvantages:
 - miscopying the code from phone to computer
 - the user usually only has 30 seconds to enter the code
 - there are not as many users that owns a smartphone, which enables a TOTP application, compared to the number of users that has a mobile phone that can receive text messages (77% of US)

Security threats:

- an attacker steals the TOTP secret from the server or the phone

Pre-generated codes

A pre-generated code is usually given to the user during registration of a two-factor authentication, see Section 2.1.4. If the user is unable to access its primary two-factor authenticator, for example, if the user loses its smartphone, the pre-generated code serves as a backup method. The user usually either prints or writes the verification code(s) down. Both the user and the server have to keep the codes secure.

Usability advantages and disadvantages:

- Advantages:
 - backup authentication mechanism for the user to recover its account
- Disadvantages:
 - a greater risk for user error when entering codes, since the codes usually are longer than SMS or TOTP codes
 - the user has to be careful not to lose the medium or device where the code is stored

Security threats:

- an attacker steals the code from the user or the server
- the codes are vulnerable to offline brute-force attacks due to long-time storage on the server

Push notifications

The user receives a push notification on their smartphone. The user can either approve or reject the login attempt.

Usability advantages and disadvantages:

- Advantages:
 - user-friendly since the user only has to press a button on the screen instead of typing a code
- Disadvantages:
 - requires Internet connection

Security threats:

- push notifications are sent to the wrong device
- communication between the user's device and the server is not secure

U2F Security Keys

Universal 2nd Factor (U2F) Security Keys is an open standard for authentication using a USB hardware device. The user connects the device to their computer and activates the device when prompted by the website.

Usability advantages and disadvantages:

- Advantage:
 - user-friendly since the user only has to connect a USB to their computer and press a button
- Disadvantage:
 - does not work with all kind of devices (differences in USB connectors/HW)

Security threats:

- the user loses its device (but this is a risk for all authenticators that has the ownership factor included)

2.1.3 Commonly used Authenticator Methods based on Biometric Factors

Authenticator methods based on biometric factors verifies an individual's identity based on their behavioural and biological characteristics, e.g., fingerprint, face recognition, and behaviour recognition [22].

A study on biometric authentication on iPhone and Android [34] states that the fingerprint scanner provides the most widely integrated biometric authenticator, since it is the most common integrated biometric interface of today's smartphones. Some of the advantages and disadvantages with fingerprint authentication via smartphones were given as a conclusion of the study:

- Advantages:
 - (security) there is a strong relationship between the user and its biometric data,
 - (usability) because people will want to use their phone in a dark area more often than when they have wet fingers, fingerprint authentication is more user-friendly than face recognition,
 - (usability) the majority of current and former users felt fingerprint unlock a little or a lot more secure than a PIN (but this is not true due to PIN being a fallback mechanism on most smartphones).
- Disadvantages:
 - (security) an attacker can intercept the information and replay it, since the information that the user provides is generally similar at each authentication attempt,

- (security) fingerprint authenticators can be fooled with molds of fingerprints,
- (usability) the fingerprint sensor did not work when the user’s fingers were wet,
- (cost) good biometric sensors are expensive.

The main security advantage of the authenticator methods based on biometric factors is that there is a strong relationship between the user and its biometric data. A security threat of biometric authentication is that an attacker can intercept the information and replay it, since the information that the user provides is generally similar at each authentication attempt. It is not recommended to use biometric authentication as a standalone authentication method [35]. There might be a cost-factor to some biometric authentication systems since the use of any biometrics often require a set of separate sensing devices. But if the sensing devices already are integrated in the users’ smartphones, the costs can be reduced.

2.1.4 Study: Account Recovery of 10 popular Web Services

To investigate how web services handle account recovery and what type of security that is in place to protect the accounts, 10 popular web service’s account recovery was studied. Some had better layers of security than others. The web services that were included in the study was:

- Email-providers:
 - Gmail
 - Yahoo
 - Microsoft’s Outlook.com
- Social networking platforms:
 - Facebook
 - Twitter
 - LinkedIn
- Tech hubs:
 - GitHub
- File Hosting Service:
 - Dropbox
- Online Retailers:
 - Amazon
 - Apple

The account recovery solution is an important part of the authentication process. If a user, for example, forgets its password/credentials or loses its device, there must be an account recovery process for it. If the account recovery process does not contain layers of security that are equivalent to the level of security the authenticator itself brings, the authenticator's level of security will be reduced to the account recovery's level of security [34]. An equivalent level of security means, in this study, using the same number of authentication factors in the account recovery process as in the authentication process, as well as using the same authentication factors for the account recovery process as for authentication process. The aim with the study is to investigate if the layer of security is equivalent to the layer of security the authenticator itself brings of 10 popular web services.

There are different types of recovery strategies: alternate email addresses and phone numbers, a form of personal identification (driver's license or passport), recovery keys or trusted devices. These strategies all have their trade-offs in *security*, *usability* (how user-friendly/complicated the process is for the user), as well as *cost* (the amount of manual support required for reviews of, for example, personal identifiers or time required for implementation of account recovery processes). Two-step verification was enabled for all web services and, if available, recovery email address was registered. Recovery phone number was registered for all web services.

Each web service was investigated as follows:

1. What type/types of recovery strategy/strategies are available if the user forgets its email address?
2. What type/types of recovery strategy/strategies are available if the user forgets its password?

The results of the study are summarized in Table 2.1.

Gmail:

Enabled two-step verification, verification code to phone number.

1. Forgot email?
 - (a) Enter your recovery phone or recovery email.
2. Forgot password?
 - (a) Tap yes on your phone or tablet.
 - (b) Send verification code to current email address (if signed in on another device).
 - (c) Send verification code to registered phone number.
 - (d) Send verification code to registered recovery email address.
 - (e) Enter the last password you remember using with this Google Account.
 - (f) Not able to recover account.

Yahoo:

Enabled two-step verification, verification code to phone number.

1. Forgot email?
 - (a) Enter your recovery phone or recovery email.
2. Forgot password?
 - (a) Send verification code to phone number.
 - (b) Verify with registered Google-account (registered recovery email address).
 - (c) Send verification code to registered recovery email address.
 - (d) all the premium support number.
 - (e) Visit the free help website (contains information about the previous verification steps).

Outlook.com:

Enabled two-step verification, verification code to phone number.

1. Forgot email?
 - (a) Enter your recovery phone or recovery email.
2. Forgot password?
 - (a) Send verification code to registered email address. After this step the user has to verify via phone number or account recovery code. If no phone number or account recovery code is given, replacing the security info of the account will take 30 days.
 - (b) Send verification code to phone number. After this step the user has to verify via recovery email address or account recovery code. If no email address or account recovery code is given, replacing the security info of the account will take 30 days.
 - (c) Enter account recovery code, possible to receive when logged in. Replacing the security info of the account will take 30 days after entering the account recovery code.
 - (d) Contact support with another email address.

Facebook:

Enabled two-step verification, verification code to phone number.

1. Forgot email?
 - (a) Enter your recovery phone or recovery email.
2. Forgot password?
 - (a) Enter full name, email address or phone number. The user will then be asked to enter its password.
 - (b) Not able to recover account.

Twitter:

Enabled two-step verification, verification code to phone number.

1. Forgot email?
 - (a) Possible to log in with username, phone number, or email address.
2. Forgot password?
 - (a) Send verification code to registered email address. After this step the user has to verify either via phone number or authentication code. Otherwise, contact support.
 - (b) Contact support.

LinkedIn:

Enabled two-step verification, verification code to phone number.

1. Forgot email?
 - (a) Possible to log in with email address or phone number.
2. Forgot password?
 - (a) Send verification code to phone number. After this step the user has to verify via email address, where a verification code is sent. If the user does not have access to the email address, a new email address can be registered for the account. But to be able to change the email address, the user has to provide a government-issued ID, which will be reviewed.
 - (b) Send verification code to email address. After this step the user has to verify by the verification code sent to the phone number. If the user don't have access to its phone, the user has to contact customer service and provide a government-issued ID.

GitHub:

Enabled two-step verification, verification code to phone number.

1. Forgot email?
 - (a) Possible to log in with email address or phone number.
2. Forgot password?
 - (a) Enter email address, a verification link is sent to the email address. The verification link redirects to a page where the user provides the verification code sent to the phone number. If the user has lost the phone, it is possible to provide the recovery codes given to the user during the enabling of 2-step verification.
 - (b) Not able to recover account.

Dropbox:

Enabled two-step verification, verification code to phone number.

1. Forgot email?
 - (a) No support, contact email provider.
2. Forgot password?
 - (a) Enter email address, a verification link is sent to the email address. The verification link redirects to a page where the user enters a new password. After this step the user is redirected to a page where the user provides the verification code sent to the phone number. If the user has lost the phone, it is possible to provide one of the recovery codes given to the user during the enabling of 2-step verification. However, even if the user does not provide the verification code sent to the phone, or one of the recovery codes, the password can be changed after verifying the email through the verification link.
 - (b) Not able to recover account.

Amazon:

Enabled two-step verification, verification code to phone number.

1. Forgot email?
 - (a) Possible to log in with email address or phone number.
2. Forgot password?
 - (a) Send verification code to the email address. The verification link redirects to a page where the user enters a new password. After this step the user is requested to log in using the new password, and after that a verification code is sent to the phone number.
 - (b) Contact customer service to recover the account.

Apple:

Enabled two-step verification, verification code to registered device.

1. Forgot email?
 - (a) Possible to log in with Apple-ID or enter first name, last name, and email address.
2. Forgot password?
 - (a) Enter Apple-ID and enter phone number. If the device is lost, the user can reset its password while signing in on a new device, use someone else's iOS device, or contact apple support and wait for them to verify the user's identity.

Web Service	Two-Step Verification for account recovery	Authentication factors for the account recovery process is the same as for the authentication process
Gmail	No	No
Yahoo	No	No
Outlook.com	Yes	No
Facebook	?	?
Twitter	Yes	Yes
LinkedIn	Yes	Yes
GitHub	Yes	Yes
Dropbox	Yes	Yes
Amazon	Yes	Yes
Apple	Yes	Yes

Table 2.1: Account recovery strategies of 10 popular web services.

During this study, it was not possible to register another email address for a Facebook account. Even though two-step verification was added for the authentication process, it was not possible to recover the account without knowing the password.

As expected, the web services that were included in the study had different levels of security in the account recovery process. Gmail only had one-step verification for the account recovery. This means that if the user's email address is compromised or the user's device is lost, the Gmail account can easily be compromised. This was also the case for Yahoo.

Outlook.com had a two-step verification for the account recovery, but did not use the same factors for the account recovery process as for the authentication process. If the user verifies via email, i.e., a verification code is sent to the user's registered email address, then it is possible to enter a account recovery code. In this study, the account recovery codes (referred to as "pre-generated codes" in Section 2.1.2) were in most cases 25 character long codes that contained characters and numbers. These codes were in most cases given during the process of enabling the two-step verification for the account. This can be seen more as an ownership factor than a knowledge factor, since it is not reasonable to remember a 25 character long randomized code and the user has to save it somewhere. Because of the fact that it was possible to replace the security info of the account (recover the account) after 30 days without the ownership factor (phone or recovery codes), a time-factor was added and the authentication factors for the account recovery process was not the same as for the authentication process. If a time-factor of 30 days achieves the same level of security as the ownership factor is not a part of this study. Therefore, there is no conclusion about whether the level of security of the account recovery process is equivalent to the level of security of the authentication process for Outlook.com accounts.

The level of security is important in the account recovery process. However, it is not user-friendly if the user is not able to recover its account if the registered devices, recovery codes, passwords or email addresses are lost or compromised. This is the case for Gmail, GitHub and Dropbox. Other web services have added manual customer support if this situation should occur, e.g., Yahoo, Outlook, Twitter, LinkedIn, Amazon and Apple. The trade-off of having the manual customer support as a part of the account recovery is an increase in usability, but might also be an increase of cost due to time and personnel.

2.1.5 Comparison of Commonly used Authenticator Methods

One of the critical drivers of designing an authentication system is the fundamental trade-off between usability and security [29]. While security is the main objective of secure systems, the user shall not be the weakest component of the authentication system. The user behaviour is an important aspect of the design of secure systems. If the user behaviour is not taken into consideration, the user's wishes of which authenticators to use for authentication might be faulty due to how much time is required to authenticate, or having too much confidence in authenticators that has a lower level of security. The cost-factor is also important when designing an authentication system. However, an authenticator that requires the user's mobile phone as a second-factor authenticator does not introduce additional costs and can be implemented relatively easy, since most users usually carry their phone all the time. Well-known companies that provide authentication that involves a mobile phone as a second-factor are Amazon, Apple, Dropbox, Google, Microsoft, Twitter, Yahoo, and many more, including a large number of financial institutions [26].

The traditional username/password authenticator is the weakest level of authentication [24], [25]. By having a minimum requirement of the complexity and length of the password, the level of security is increased, but it is still lower than the authenticator methods based on ownership factors and biometric factors. The ownership factors presented in Section 2.1.2 are all two-factor authentication methods, since they all include something that the user has and something that the user knows. The fingerprint authenticator has the strongest relationship between the user and its biometric data. Therefore, the username/password authenticator should be considered as low in security but high in usability. The user only needs to remember a password, a passphrase, or a PIN, and therefore, this authentication method is one of the most user-friendly.

A comparison of the subjective time it takes for a user to authenticate with the authentication methods used in the study presented in Section 2.1.2 is shown in Table 2.2.

Authentication Method	Authentication Time (1-5), 1=fastest, 5=slowest
SMS	3
TOTP	4
Pre-generated codes	5
Push notifications	2
U2F Security Keys	1

Table 2.2: Authentication time for common authenticator methods [33].

The pre-generated codes are, according to the study, the authentication method that requires most authentication time for the user, i.e., the authentication method that is least user-friendly when looking at authentication time. There is also a greater risk for user-error when the user enters the codes since they usually are longer than SMS or TOTP codes. Moreover, there are security risks with storing a code for a long time period (which is required in this method) on a server, due to offline brute-force attacks or if an attacker steals the code from the user or the server in another way. The other authenticators that requires secret or code storage are SMS, TOTP and U2F security keys. Even if the TOTP authentication method requires more time than the SMS authentication method, it has a higher level of security, since the SMS authenticator might be exposed to man-in-the-middle-attacks (SIM-swapping attacks [43]), brute force attacks and targeted phishing attacks. Even if the TOTP secret is hashed on the server, it can be stolen by an attacker from the server or the phone (as for the SMS authenticator code), but the TOTP authenticator does not involve sending any codes via SMS or to the server.

The push notifications and the U2F security keys are more user-friendly when it comes to authentication time. The U2F security key is an open standard for authentication using a USB hardware device. To compare the push notification's and security key's level of security, a formal security analysis and risk analysis of push-based authentication has to be made to define its security vulnerabilities and threats (this is out of scope for this Thesis). The U2F security keys are designed to be both highly secure and usable, but it might be costly to provide each user with a security key.

There is no authentication factor in any other authenticator that has a stronger relationship to the user than a biometric factor in the biometric authenticators. The use of a biometric factor has the potential to bring a high level of security. The usability of a fingerprint authentication process can be compared to the push notification authenticator, since it does not require much more effort from the user than holding a fingerprint sensor on the phone instead of pressing a button on the screen on the phone. Therefore, the authentication time for a fingerprint authen-

enticator will not differ much from the authentication time for a push notification authenticator. However, there are security risks with biometric factors that leaves a local device since an attacker can intercept the biometric information and replay it. There is also a cost-factor that depends on if the sensing devices are integrated in the users' smartphones or not. Otherwise, there might be increase costs, since the use of any biometrics often require a set of separate sensing devices. But the most important thing to consider when it comes to security of the biometric authenticator is that it does not have a higher level of security than its fallback authenticator, in the case where the fallback method is less secure. This means that if the user loses its smartphone, another sensing device that can process biometric data must take its place to maintain the level of security that biometric authentication brings.

The overall conclusions of this chapter that are used in the results of this project are:

- The traditional username/password authenticator is low in security but high in usability. It is one of the most user-friendly authentication methods based on the knowledge factor.
- The SMS authentication method might requires less authentication time for the user than the TOTP authentication method, but the TOTP method has a higher level of security.
- If push-notification authentication is used, a formal security analysis and risk analysis should be made.
- Using U2F security keys is both highly secure and usable, but it is costly to provide each user with a security key. To reduce costs, another authentication method based on the ownership factor is user authentication with something that the user already has, e.g., a smartphone.
- There is no authentication factor in any other authenticator that has a stronger relationship to the user than a biometric factor. Moreover, if the user owns a smartphone with a biometric sensor, costs can be reduced.
- If biometric data is used in an authentication process, there are security risks and privacy concerns with biometric factors that leaves a local device or is stored in a database. In other words, biometric data should not be stored or leave any local device.

The OAuth 2.0 Authorization Framework, RFC6749

The OpenID Connect protocol is a widely used protocol that is an extension to OAuth 2.0. To be able to easily integrate a system with another system (if needed), there are benefits with using industry-standard protocols for authorization and authentication, i.e., OAuth 2.0 and OpenID Connect. Inventing a protocol on your own is a possibility. However, research, development, and a lot of effort has been put into these protocols to define everything between security vulnerabilities to requirements for a client. In order to gain in-depth knowledge of the OpenID Connect protocol, in-depth knowledge for OAuth 2.0 is also required. This chapter gives a summary of the flows and the basics of the OAuth 2.0 Authorization Framework [3]. For the thesis implementation work it was important to truly understand the OAuth protocol.

The Internet Engineering Task Force (IETF) is a large open international organization that writes specifications that standardize different parts of the Internet to make the Internet work better. IETF is a community of network designers, operators, vendors and researchers concerned with the evolution of the Internet architecture and the smooth operation of the Internet. [1]

An RFC, Request For Comments, is a document or a document series that contains technical and organizational notes about the Internet, e.g., protocols, research, standards, programs and concepts. The RFCs are published with approval from the IETF. [2]

The OAuth 2.0 Authorization Framework is an Internet Standards Track document, a protocol that enables a third-party application to obtain limited access to an HTTP service.

In traditional client-server authentication models, the client authenticates with the server using the resource owner's credentials. In order for third-party applications to get access to protected resources, the resource owner shares its credentials with the third-party. There are several problems and limitations with this:

- The third-party application stores the resource owner's credentials for future

use, e.g., resource owner's password in clear text.

- The servers are required to support authentication, e.g., password authentication.
- The resource owner is unable to limit the scope (see below) of the resources that are accessed by the third-party application.
- The resource owner can only remove the access that one third-party application has by removing the access that all third-party applications have, by changing the third party's password.
- If the third-party application is compromised, the third-party's password is compromised, and the resources that are protected by the password can be exposed.

In OAuth, the "scope" parameter is used by an app to specify what access is needed, e.g., user information, such as home address. The authorization server (see below) uses the "scope" parameter to respond with the access that was granted (if the granted access is different from the requested access). The OAuth 2.0 introduces an authorization layer, where the client requests access to resources that are controlled by the resource owner and hosted by the resource server, and is issued a different set of credentials than those of the resource owner. There are four roles defined in this protocol:

- **Resource Owner:** If the resource owner is a person, it is referred to as an end-user. The resource owner is capable of granting access to a protected resource.
- **Resource Server:** The resource server accepts or denies resource requests that contains access tokens.
- **Client:** The client is an application that makes protected resource requests on behalf of the resource owner with its authentication and authorization. See more information about clients in Section 3.1.
- **Authorization Server:** The authorization server issues the access tokens to the client after authenticating the resource owner and obtaining authorization.

The resource server and the authorization server can, for example, be the same server but exist as different endpoints on this server.

An authorization grant expresses the authorization that the client uses to request an access token or a refresh token. The authorization is obtained from the resource owner. The OAuth 2.0 protocol describes four different grant types, i.e., four protocol flows:

- Authorization Code Grant
- Implicit Grant
- Resource Owner Password Credentials Grant
- Client Credentials Grant

(The protocol also describes an extensibility mechanism for defining additional types, which is out of scope for this Thesis.)

3.1 Client Types, Profiles, Registration, Authentication and Identifier

The two different **client types**, defined by the OAuth 2.0 protocol are confidential clients and public clients. A *confidential client* is capable of maintaining confidential information (requires a trusted back-end server to store the secret(s)) or able to securely authenticate with the authorization server. *Public clients* are not capable of maintaining confidential information (e.g, client is executing on a web-browser based application) and are not able to securely authenticate.

The OAuth 2.0 protocol is designed around three different **client profiles**: web application, user-agent-based application and native application. The *web application* is a confidential client running on a web server. Client credentials, tokens and other confidential information are stored on the web server, i.e., confidential information is not exposed to the resource owner. The *user-agent-based application* is a public client, where the client code is downloaded from a web server and executes on a user agent (e.g., web browser) on the device that is used by the resource owner. Client credentials are in this case visible/accessible by the resource owner. A *native application* is a public client that is installed and executed on the device used by the resource owner. Client credentials and protocol data are visible/accessible by the resource owner, but are protected from other servers and applications.

The OAuth 2.0 protocol does not specify how the client should register with the authorization server, but it typically involves end-user interaction with an HTML registration form. The main purpose of **client registration** is to establish a trust relation and obtain the client properties (e.g., redirection URI and client type) that are required by the authorization server. Registration can, for example, be accomplished using a self-issued or third-party-issued assertion, or by the authorization server performing client discovery using a trusted channel. When the client is registered, the client developer shall specify the client type, provide the client redirection URIs and include any other information that is required by the authorization server. Registration is a crucial process and depending on the use case, and if the user interaction and vetting process is very involved or not. We do not consider the registration process in this thesis.

Client authentication for confidential clients can be done with any form or method that is meeting the authorization server's security requirements. The confidential clients are usually issued a set of client credentials used for authenticating with the authorization server (e.g., client password). Client authentication for public clients may be established with the authorization server, but the authorization server must not rely on public client authentication for the purpose of identifying the client. Confidential clients or other clients that are issued client

credentials must authenticate with the authorization server when making requests to the token endpoint. Client authentication is used for:

- Refresh tokens, see Section 3.3.1, and authorization codes are bound to the client they were issued to. The client authentication is extra important when the authorization code is issued over an insecure channel or when the redirection URI has not been registered in full.
- It is more efficient to change a single set of client credentials than revoking an entire set of refresh tokens, in the case where a client gets compromised, i.e., preventing an attacker from abusing stolen refresh tokens (they can not be used without the client credentials or identifier).
- Periodic credential rotation is recommended when implementing authentication management best practices. It is also in this case easier to rotate a single set of client credentials, instead of rotating an entire set of refresh tokens.

The **client identifier** is a unique string that is representing the registration information provided by the client. The client identifier is issued by the authorization server and is unique to the server. The identifier is not a secret and it is exposed to the resource owner. Therefore, the client identifier must not be used alone for client authentication.

3.2 Protocol Endpoints

There are three endpoints defined by the OAuth 2.0 protocol: the authorization endpoint, the token endpoint and the redirection endpoint. The two authorization server endpoints (HTTP resources) are the *authorization endpoint* and the *token endpoint*, while the *redirection endpoint* is the client endpoint. Both the authorization server endpoints are not used by all the authorization grants defined in this protocol.

3.3 Tokens

This section describes the tokens that are used in the OAuth 2.0 protocol to access protected resources. The two tokens defined in the protocol are access tokens and refresh tokens.

3.3.1 Access Tokens and Refresh Tokens

An access token is a credential that is used to access protected resources. It is an opaque string or a JSON web token that is issued to the client, which is representing an authorization instead of using different authorization constructs, e.g., username and password. The tokens can be generated using different algorithms, have different life-time and size, have different scopes of access, and have cryptographic properties.

A refresh token is a credential that is used to obtain new or additional access tokens from the authorization server. An access token may have expired or a different scope (than the current access token has) might be requested by the client. In both cases, the client can send a refresh token to the authorization server to obtain a new access token. Only the access tokens, and not the refresh tokens, are sent to the resource server.

3.3.2 Issuing an Access Token

This section describes the successful response and the error response to an access token request.

3.3.3 Successful Response

An access token and, optionally, a refresh token is sent from the authorization server to the client if the access token request is valid and authorized. The following parameters are included in the access token response:

```
//REQUIRED PARAMETERS:
access_token
    The access token
token_type
    The type of the token that is issued by the
    authorization server, examples of access token types
    are "bearer" and "mac"
//RECOMMENDED PARAMETERS:
expires_in
    The access token's lifetime in seconds, from the
    time that the response was generated
//OPTIONAL PARAMETERS:
refresh_token
    The refresh token
scope //REQUIRED IF PRESENT IN REQUEST
    A list of space-delimited, case-sensitive strings
    that specifies the scope of the request
```

The "application/json" media type is used for the HTTP response, which includes the parameters above. The parameters are included as JSON strings and JSON numbers into a JSON object. The HTTP "Cache-Control" with the value "no-store" and "Pragma" with the value "no-cache" must be included in the response. An example of a successful access token response:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "access_token": "2YotnFZFEjr1zCsicMWpAA",
```

```

    "token_type":" example " ,
    "expires_in":3600 ,
    "refresh_token ":"tGzv3JOkF0XG5Qx2TIKWIA " ,
    "example_parameter ":" example_value "
}

```

3.3.4 Error Response

The following parameters are included in the error response:

```

//REQUIRED PARAMETERS
error
    An ASCII error code form the following:
        invalid_request
        invalid_client
        invalid_grant
        unauthorized_client
        unsupported_grant_type
        invalid_scope
//OPTIONAL PARAMETERS
error_description
    Human-readable ASCII with additional information
error_uri
    URI to a web page with human-readable information about
    the error

```

The "application/json" media type is used for the HTTP response, which includes the parameters above. The parameters are included as JSON strings and JSON numbers into a JSON object. The HTTP "Cache-Control" with the value "no-store" and "Pragma" with the value "no-cache" must be included in the response. An example of an error response:

```

HTTP/1.1 400 Bad Request
Content-Type: application/json; charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

```

```

{
    "error":"invalid_request"
}

```

3.4 HTTP Redirections

The protocol uses HTTP 302 status code for HTTP redirections. However, other methods are allowed and are considered to be implementation details. The HTTP redirections in the different OAuth 2.0 flows are used to redirect the resource owner's user-agent to other destinations. The redirections are made by the client or the authorization server.

3.5 Authorization Code Flow

This section describes the authorization code flow of the OAuth 2.0 protocol. The purpose of the messages in Figure 3.1 are briefly explained below, and subsequently we describe messages 1, 3, 4, and 5 in more detail in Appendix A.1. OAuth 2.0 is not an *authentication protocol*. Therefore, the user authentication, i.e., message 2, is not explained in detail in this OAuth 2.0 summary.

- Obtain both access tokens and refresh tokens
- Optimized for confidential clients
- Redirection-based flow, see Section 3.4, which means that the client must be capable of interacting with the resource owner's user-agent and receiving incoming requests from the authorization server (via redirection)

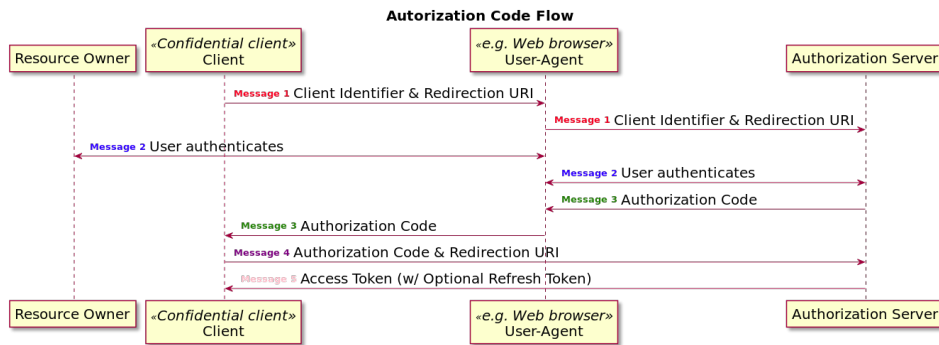


Figure 3.1: An overview of the authorization code flow.

Message 1: Client directs the resource owner's user-agent to the authorization endpoint. Client identifier, requested scope, local state and redirection URI is included. See Section A.1.1 for more details.

Message 2: The authorization server authenticates the resource owner via the user-agent and determines if the resource owner grants or denies the client's access request.

Message 3: If the resource owner grants access, the authorization server redirects the user-agent back to client (using the redirection URI that also includes an authorization code and any local state from message 1). See Section A.1.2 and A.1.3 for more details.

Message 4: Client authenticates with the authorization server and requests an access token from the authorization server's token endpoint. The request includes the redirection URI from message 1 and the authorization code from message 3. See Section A.1.4 for more details.

Message 5: The authorization server authenticates the client, validates the authorization code and ensure that the redirection URI is the same as the one in message 3. If the client is authenticated and the code and the redirection URI is valid, the authorization server responds with an access token and (optionally) a refresh token. See Section A.1.5 for more details.

3.6 Implicit Flow

This section describes the implicit flow of the OAuth 2.0 protocol. This flow is optimized for public clients, compared to the authorization code flow that is optimized for confidential clients. The purpose of the messages in Figure 3.2 are briefly explained below, and subsequently we describe messages 1 and 3 in more detail in Appendix A.2.

- Obtain access tokens
- Optimized for public clients, known to operate at a particular redirection URI
- Redirection-based flow, see Section 3.4, which means that the client must be capable of interacting with the resource owner's user-agent and receiving incoming requests from the authorization server (via redirection)
- Client receives token response as a result of the authorization request (the authorization server only has one endpoint in this case)
- No client authentication is performed by the authorization server in this flow.
- Relies on the presence of the resource owner and the registration of the redirection URI. The access token is encoded into the redirection URI. Therefore, the access token may be exposed to the resource owner and other applications on the same device.

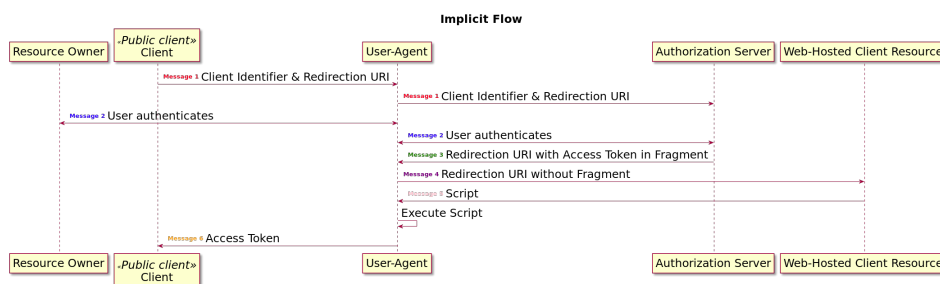


Figure 3.2: An overview of the implicit flow.

Message 1: Client directs the resource owner's user-agent to the authorization endpoint. Client identifier, requested scope, local state and redirection URI is included. See Section A.2.1 for more details.

Message 2: The authorization server authenticates the resource owner via the user-agent and determines if the resource owner grants or denies the client's access request.

Message 3: If the resource owner grants access, the authorization server redirects the user-agent back to client (using the redirection URI that also includes the access token in the URI fragment). See Section A.2.2 for more details. Otherwise, an error response is sent (see Section A.2.3).

Message 4: The user-agent makes a request to the web-hosted client resource by following the redirection instructions. The user-agent retains the fragment information locally, and does not include the fragment in the request to the web-hosted client resource.

Message 5: The web-hosted client resource returns a web page that can access the full redirection URI, as well as the user-agent's locally retained fragment (e.g., extraction of access token is possible).

Execute Script: The user-agent executes the script received from the web-hosted client resource locally, which extracts the access token.

Message 6: The user-agent passes the access token to the client.

3.7 Resource Owner Password Credentials Flow

This section describes the resource owner password credentials flow of the OAuth 2.0 protocol. The purpose of the messages in Figure 3.3 are briefly explained below, and subsequently we describe all messages in more detail in Appendix A.3.

- Suitable when the resource owner has a trust relationship with the client (e.g., device operating system or a highly privileged application)
- This flow should only be used when other flows are not viable
- Suitable for clients that are able to obtain the resource owner's credentials (username and password, using an interactive form)
- Also used to migrate existing clients

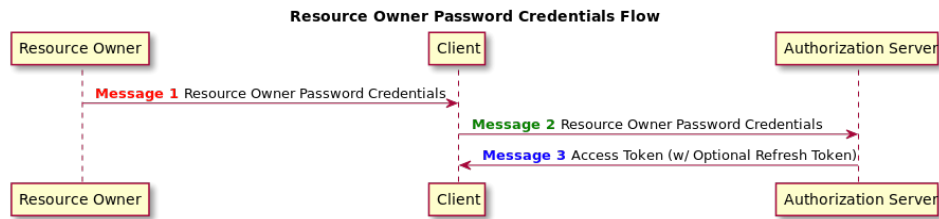


Figure 3.3: An overview of the resource owner password credentials flow.

Message 1: The resource owner sends its username and password to the client. See Section A.3.1 for more details.

Message 2: The client sends the username and password, received in message 1, to the token endpoint at the authorization server to request an access token. The client also authenticates with the authorization server when making the request. See Section A.3.2 for more details.

Message 3: The authorization server authenticates the client, validates the resource owner credentials and issues an access token if client is authenticated and the credentials are valid. See Section A.3.3 for more details.

3.8 Client Credentials Flow

This section describes the client credentials flow of the OAuth 2.0 protocol.

In this flow, the client uses its credentials to request an access token. This means that the client authentication is used as the authorization grant, i.e., no additional authorization request is needed.

The purpose of the messages in Figure 3.4 are briefly explained below, and subsequently we describe all messages in more detail in Appendix A.4.



Figure 3.4: An overview of the client credentials flow.

Message 1: The client authenticates with the authorization server and requests an access token from the token endpoint at the authorization server. See Section

A.4.1 for more details.

Message 2: The authorization server authenticates the client, and issues an access token if the client authentication is valid. See Section A.4.2 for more details.

OpenID Connect Protocol

This chapter is a summary of OpenID Connect, which is an identity layer on top of the OAuth 2.0 protocol [8]. This part describes the differences and additions of OpenID Connect to the OAuth 2.0 protocol. The description of the flows of the OAuth 2.0 protocol can be found in Chapter 3.

OpenID Connect enables clients to verify the identity of the end-user based on authentication performed by an authorization sever. An ID token is a primary extension to the OAuth 2.0 protocol, and the ID token extends the OAuth 2.0 protocol to also include authentication of end-users, see Section 4.5. OpenID Connect also enables clients to obtain basic profile information about the end-user by using claims (in an interoperable and REST-like manner).

4.1 Terminology

The following terms are defined by the OpenID Connect specification:

Authentication Context: A client (RP, see below) can require an authentication context, i.e., information, before making a decision about an authentication response. The information can, for example, include the authentication method used.

Authentication Context Class: A set of authentication methods or procedures that are considered to be equivalent in a specific context.

Authentication Context Class Reference: Identifier of an authentication context class.

Claim: A piece of information that is asserted about an entity, e.g., an end-user.

OpenID Provider (OP): An authorization server, defined in the OAuth 2.0 protocol, that also is capable of authenticating the end-user and providing claims to a client (RP) about the authentication event and the end-user.

Relying Party (RP): A client application, defined in the OAuth 2.0 protocol, that also requires end-user authentication and claims from an authorization server (OP).

Subject Identifier: A locally unique and never reassigned identifier within the issuer (the entity that issues a set of claims) for the end-user, which is intended to be consumed by the client.

4.2 Authentication

The process of authentication is used to achieve sufficient confidence in the binding between the entity (e.g., an end-user) and the presented identity (i.e., value that uniquely characterizes an entity in a specific context). The result of the authentication defined by this protocol is the ID Token, see Section 4.5, and the authentication is performed to log in the end-user or to determine that the end-user already is logged in. The result of the authentication (the ID token) that is performed by the server is returned to the client in a secure manner, so that the client can rely on it. This is the reason why the client is called the relying party (RP) in this protocol.

The flows that are defined and used in the OpenID Connect protocol are the authorization code flow, the implicit flow, and the hybrid flow. The flows determine how the ID token and the access token are returned to the client. Table 4.1 gives a summary of the three flows, and is a guidance to help choose which flow to use depending on context.

Property	Authorization Code Flow	Implicit Flow	Hybrid Flow
All tokens returned from Authorization Endpoint	no	yes	no
All tokens returned from Token Endpoint	yes	no	no
Tokens not revealed to User Agent	yes	no	no
Client can be authenticated	yes	no	yes
Refresh Token possible	yes	no	yes
Communication in one round trip	no	yes	no
Most communication server-to-server	yes	no	varies

Table 4.1: OpenID Connect Authentication Flows

4.3 Requesting Claims using Scope Values

Claims can be requested using scope values, which specifies what access privileges are being requested for access tokens, i.e., what resources will be available when they are used to access OAuth 2.0 protected endpoints. Multiple scopes may be requested by creating a space delimited, case sensitive list of ASCII scope values. The following scope values are defined by the OpenID Connect specification to request claims:

```
//OPTIONAL VALUES:
```

```
profile
```

```
    The end-user's default profile claims ("name",
```

```
"family_name", "given_name", "middle_name",
"nickname", "preferred_username", "profile",
"picture", "website", "gender", "birthdate",
"zoneinfo", "locale", and "updated_at")
email
    The "email" and the "email_verified" claims
address
    The "address" claim
phone
    The "phone_number" and "phone_number_verified"
    claims
offline_access
    Requests that an OAuth 2.0 refresh token is issued
    that can be used to obtain an access token that
    grants access to the end-user's userinfo endpoint,
    even if the end-user is not logged in
```

The claims of the scope values profile, email, address and phone are returned from the UserInfo Endpoint, see Section 4.6. But when no access token is issued (e.g., when the "response_type" has the value "id_token"), the claims are returned in the ID token.

A "prompt" parameter with the value "consent" must be present for permitting offline access (unless specified otherwise). The authorization server (OP) must always obtain a consent when returning a refresh token that gives offline access, since it is not always sufficient with a previous consent. The authorization server (OP) must also: ignore the "offline_access" request if "response_type" value is "code", receive or have consent for all clients (RPs) when the "application_type" is "web" (the authorization server (OP) should receive or have consent for all clients (RPs) when the "application_type" is "native").

Example of a scope request:

```
scope=openid profile email phone
```

4.4 Client Authentication

The following client authentication methods are defined for clients to authenticate with the authorization server when using the token endpoint (if no client method is registered, the default method is "client_secret_basic"):

```
client_secret_basic
    Client has a client_secret value from the
    authorization server, authenticates with the
    authorization server using the HTTP basic
    authentication scheme
client_secret_post
    Client has a client_secret value from the
```


authorization server, authenticates with the authorization server by including the client credentials in the request body

client_secret_jwt

Client has a `client_secret` value from the authorization server, authenticates with the authorization server by creating a JWT (using HMAC SHA algorithm, e.g., HMAC SHA-256, by using the octets of the UTF-9 representation of the `client_secret` as the shared key), the following claims are included in the JWT:

```
//REQUIRED CLAIMS:
iss
sub
aud
jti
exp
//OPTIONAL CLAIMS:
iat
```

The authentication token must be sent as the value of the `"client_assertion"` parameter, and the value of the `"client_assertion_type"` must be

```
"urn:ietf:params:oauth:client-assertion-type:jwt-bearer"
```

See JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants [13] and Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants [14] for more information.

private_key_jwt

Client has registered a public key, the JWT is signed using that key, the following claims are included in the JWT:

```
//REQUIRED CLAIMS:
iss
sub
aud
jti
exp
//OPTIONAL CLAIMS:
iat
```

The authentication token must be sent as the value of the `"client_assertion"` parameter, and the value of the `"client_assertion_type"` must be

```
"urn:ietf:params:oauth:client-assertion-type:jwt-bearer". See JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization
```

Grants [13] and Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants [14] for more information.

The following is an example of a token request, a HTTP POST, involving client authentication, sent by the client to the token endpoint:

```
POST /token HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded
```

```
grant_type=authorization_code&
code=i1WsRn1uB1&
client_id=s6BhdRkqt3&
client_assertion_type=urn%3Aietf%3Aparams
%3Aoauth%3Aclient-assertion-type
%3Ajwt-bearer&
client_assertion=PHNhbWxwOl ... ZT
```

none

Client does not authenticate with the token endpoint, which might be the case in the implicit flow (the token endpoint is not used in this flow) or for a public client (no client credentials)

4.5 ID Token

An ID token is a primary extension to the OAuth 2.0 protocol. The ID token is represented as a JSON Web Token (JWT) and is a security token that contains claims about the authentication performed by an authorization server of an end-user, when the end-user is using a client. In other words, the ID token extends the OAuth 2.0 protocol to also include authentication of end-users. The following claims are defined in the OpenID Connect, which are used within the ID Token:

//REQUIRED CLAIMS:

iss

A case sensitive URL that uses the https scheme, containing scheme, host, and optionally, port number and path components, and is the issuer identifier for the issuer of the response

sub

A case sensitive string that is the subject identifier, see Section 4.1

aud

A single case sensitive string that defines the audience that the ID token is intended for, must contain the OAuth 2.0 "client_id" of the client (RP

```

    ), may also contain other identifiers for audiences
    (an array of case sensitive strings)
exp
    JSON number representing the expiration time (in
    seconds) on or after which the ID token must not be
    accepted for processing
iat
    JSON number representing the time (in seconds) at which
    the JWT was issued
//OPTIONAL CLAIMS:
auth_time //REQUIRED IF "max_age" OR "auth_time" IS PRESENT
    IN REQUEST
    JSON number representing the time (in seconds) at which
    the end-user authentication occurred
nonce //REQUIRED IF PRESENT IN REQUEST
    A case sensitive string that associates a client
    session with an ID token to mitigate replay attacks
    , passed on unmodified from the authentication
    request
acr
    A case sensitive string that is the authentication
    context class reference , where value "0" means that
    the authentication of the end-user did not meet
    the requirements (specified in ISO/IEC 29115 [10])
    for level 1, "level 0" authentication can, for
    example, be appropriate for authentication using a
    long-lived browser cookie but not appropriate for
    authorization of any resource of monetary value
amr
    JSON array of case sensitive strings that are the
    authentication methods references , i.e., the
    authentication methods used in the authentication
azp
    A case sensitive string containing a StringOrURI value
    that represents the authorized party to which the
    ID token was issued , it must contain the OAuth 2.0
    client ID of this party , the claim is only needed
    when the ID token has a single audience value and
    that audience is different than the authorized
    party

```

The ID tokens must be signed using JSON Web Signature (JWS) and, optionally, also encrypted using JSON Web Encryption (JWE). If the ID token is both signed and encrypted, it is referred to as a Nested JWT and is providing authentication, integrity, non-repudiation and confidentiality. Unless the response type returns no ID token from the authorization endpoint (or the client explicitly requested the use of "none" at registration time), the id tokens must not use the "alg" value "none". The references to keys used are communicated in advance using (the au-

thorization server's (OP's) discovery and registration parameters.

An example of a set of claims in an ID token:

```
{
  "iss": "https://server.example.com",
  "sub": "24400320",
  "aud": "s6BhdRkqt3",
  "nonce": "n-0S6_WzA2Mj",
  "exp": 1311281970,
  "iat": 1311280970,
  "auth_time": 1311280969,
  "acr": "urn:mace:incommon:iap:silver"
}
```

4.6 UserInfo Endpoint

The userinfo endpoint at the authorization server is an OAuth 2.0 protected resource that returns claims to the client (RP) about the authenticated end-user, in response to a userinfo request with an access token included. The claims in the userinfo response are usually name/value-pairs of a JSON object. All communication with the userinfo endpoint must utilize TLS. The userinfo endpoint must support the HTTP GET and HTTP POST methods, and accept access tokens as bearer tokens (see OAuth 2.0 Bearer Token Usage [11]).

4.6.1 UserInfo Request

A userinfo request is sent from the client (RP) to the userinfo endpoint at the authorization server by using the HTTP GET or the HTTP POST method. However, the HTTP GET method is recommended, as well as the access token being sent using the authorization header field. The access token must be sent as a bearer token.

An example of a userinfo request:

```
GET /userinfo HTTP/1.1
Host: server.example.com
Authorization: Bearer SlAV32hkKG
```

4.6.2 UserInfo Response

A userinfo response is sent from the userinfo endpoint at the authorization server to the client (RP). If the userinfo request sent does not create an error condition, a successful userinfo response is sent (see Section 4.6.2). Otherwise, an authentication error response is sent (see Section 4.6.2).

Successful UserInfo Response

If the userinfo response body is a text JSON object, the "application/json" format must be used for the userinfo response, sent from the userinfo endpoint to the client (RP). If the userinfo response is signed and/or encrypted, the "application/jwt" format must be used for the userinfo response, since the claims are returned in a JWT in this case. The userinfo endpoint must return a content-type header to indicate which format is being used. If the userinfo response is signed, it should contain the "iss" (authorization server's (OP's) issuer identifier URL) and "aud" (client id value) claims.

The userinfo response includes the claims that were included in the userinfo request, except for values that the authorization server (OP) may elect due to privacy reasons. The "sub" claim must always be returned in the userinfo response.

Example of a userinfo response, a HTTP response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "sub": "248289761001",
  "name": "Jane Doe",
  "given_name": "Jane",
  "family_name": "Doe",
  "preferred_username": "j.doe",
  "email": "janedoe@example.com",
  "picture": "http://example.com/janedoe/me.jpg"
}
```

Userinfo Error Response

If an error condition occurs, the userinfo endpoint at the authorization server will send a userinfo error response to the client (RP), defined in Section 3 of OAuth 2.0 Bearer Token Usage [12].

Example of a userinfo error response, a HTTP response:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: error="invalid_token",
  error_description="The Access Token expired"
```

Userinfo Response Validation

The client (RP) must verify that the authorization server (OP) was the intended server through a TLS certificate check. If the "userinfo_encrypted_response_alg" parameter is provided by the client (RP) during registration, the client must decrypt the response using the keys that are spec-

ified during registration. The client should validate the signature according to JWS if the userinfo response was signed.

4.7 Authorization Code Flow

This section describes the authentication using the authorization code flow. The purpose of the messages in Figure 4.1 are briefly explained below, and subsequently, we describe the messages, the endpoints, and the ID and access tokens in more detail in Appendix B.1.

- All tokens are returned from the token endpoint.
- The authorization endpoint returns an authorization code to the client, which can be exchanged for an ID token and an access token at the token endpoint. Hence, no tokens are exposed to the user-agent or other malicious applications with access to the user-agent.
- The authorization server can authenticate the client before exchanging the authorization code for an access token.
- Suitable for confidential clients.

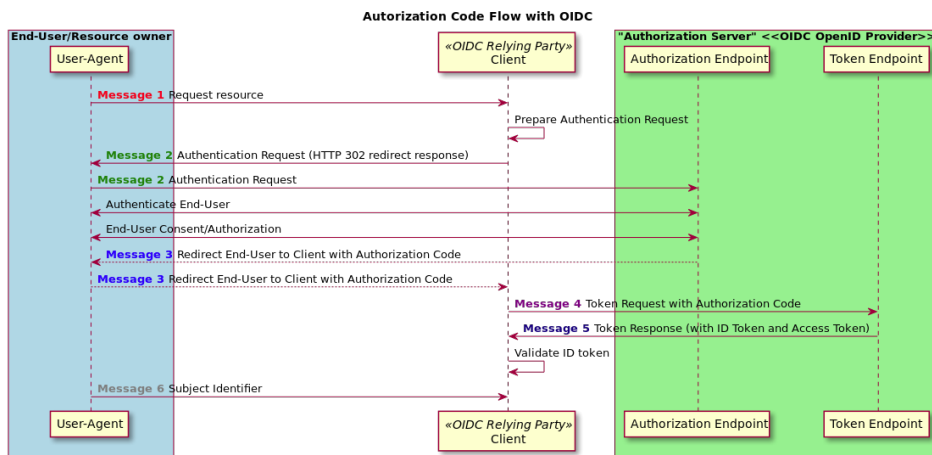


Figure 4.1: An overview of Authentication using the Authorization Code Flow.

Message 1: The end-user sends a resource request to the client.

Prepare Authentication Request: The client prepares the authentication request with the desired parameters. See Section B.1.4 for more details.

Message 2: The client sends the authentication request to the authorization endpoint at the authorization server. See Section B.1.4 for more details.

Authenticate End-User: The authorization server authenticates the end-user. See Section B.1.5 for more details.

End-User Consent/Authorization: The authorization server obtains the end-user consent/authorization. See Section B.1.6 for more details.

Message 3: The authorization server sends the end-user back to the client with the authorization code. See Section B.1.7 for more details.

Message 4: The client sends a token request to the token endpoint at the authorization server using the authorization code. See Section B.1.8 for more details.

Message 5: The authorization server sends a token response to the client containing an ID token and an access token (and optionally a refresh token). See Section B.1.9 for more details.

Validate Token: The client validates the token. See Section B.1.9 for more details.

Message 6: The client retrieves the end-user's subject identifier.

4.8 Implicit Flow

This section describes the authentication using the implicit flow. The purpose of the messages in Figure 4.2 are briefly explained below, and subsequently, we describe the messages, the endpoints, and the ID and access tokens in more detail in Appendix B.2.

- All tokens are returned from the authorization endpoint, the token endpoint is not used.
- Clients are usually implemented in a browser using a scripting language.
- Access tokens and ID tokens might be exposed to end-users and applications that have access to the end-user's user-agent, since the access tokens and id tokens are returned directly to the client.
- The authorization server does not perform client authentication.

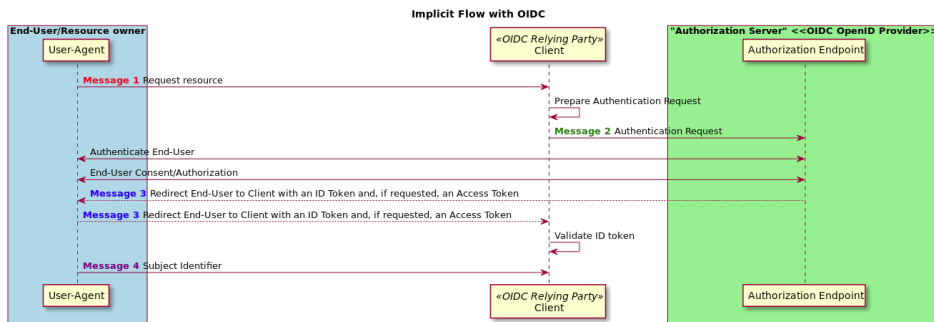


Figure 4.2: An overview of Authentication using the Implicit Flow.

Message 1: The end-user sends a resource request to the client.

Prepare Authentication Request: The client prepares the authentication request with the desired parameters. See Section B.2.4 for more details.

Message 2: The client sends the authentication request to the authorization endpoint at the authorization server. See Section B.2.4 for more details.

Authenticate End-User: The authorization server authenticates the end-user. See Section B.2.5 for more details.

End-User Consent/Authorization: The authorization server obtains the end-user consent/authorization. See Section B.2.6 for more details.

Message 3: The authorization server sends the end-user back to the client with the ID token and, if requested, an access token. See Section B.2.7 for more details.

Validate Token: The client validates the ID token and, if requested, an access token. See Section B.2.7 for more details.

Message 4: The client retrieves the end-user's subject identifier.

4.9 Hybrid Flow

This section describes the authentication using the hybrid flow. The purpose of the messages in Figure 4.3 are briefly explained below, and subsequently, we describe the messages, the endpoints, and the ID and access tokens in more detail in Appendix B.3.

- Tokens are returned both from the authorization endpoint and the token endpoint.

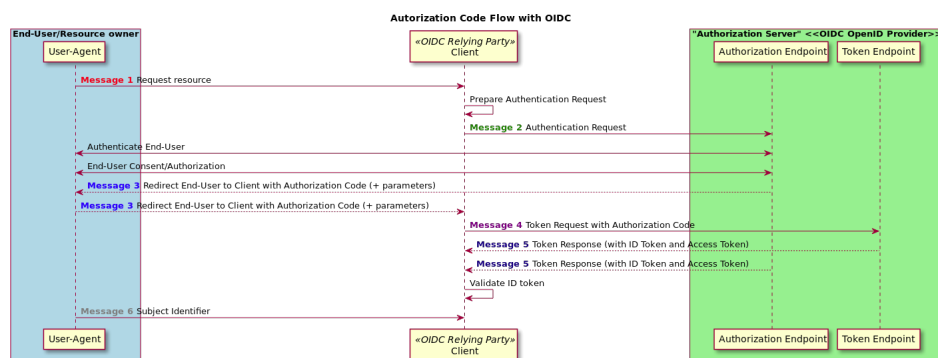


Figure 4.3: An overview of Authentication using the Hybrid Flow.

Message 1: The end-user sends a resource request to the client.

Prepare Authentication Request: The client prepares the authentication request with the desired parameters. See Section B.3.5 for more details.

Message 2: The client sends the authentication request to the authorization endpoint at the authorization server. See Section B.3.5 for more details.

Authenticate End-User: The authorization server authenticates the end-user. See Section B.3.6 for more details.

End-User Consent/Authorization: The authorization server obtains the end-user consent/authorization. See Section B.3.7 for more details.

Message 3: The authorization server sends the end-User back to the client with the authorization code and one or more parameters (depending on the response type). See Section B.3.8 for more details.

Message 4: The client sends a token request to the token endpoint at the authorization server using the authorization code. See Section B.3.9 for more details.

Message 5: The authorization server sends a token response to the client containing an ID token and an access token (and optionally a refresh token). The tokens are sent either from the authorization endpoint, the token endpoint or both the authorization endpoint and the token endpoint. See Section B.3.10 for more details.

Validate Token: The client validates the token. See Section B.3.2 and B.3.3 for more details.

Message 6: The client retrieves the end-user's subject identifier.

Proof Key for Code Exchange by OAuth Public Clients

This chapter is a summary of a specification that specifies an attack that OAuth 2.0 public clients are exposed to, for which the technique Proof Key for Code Exchange (PKCE) provides a mitigation [9].

There is a security vulnerability in the OAuth 2.0 protocol for public clients in the authorization code flow, see Section 3.5, that can be mitigated by dynamically creating a cryptographically random key called "code verifier". The PKCE specification adds additional parameters to some requests and responses in the authorization code flow, to prevent an attacker from intercepting the authorization code in the authorization code flow.

5.1 Authorization Code Interception Attack Flow

This section describes how an interception attack can occur in the OAuth 2.0 authorization code flow.

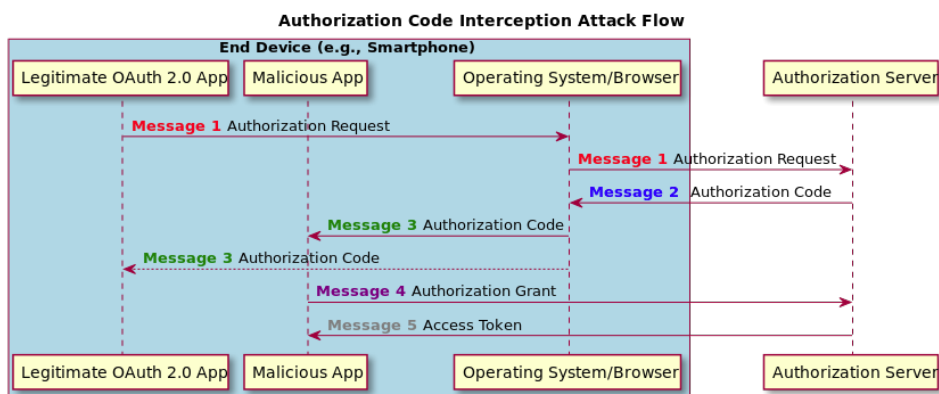


Figure 5.1: An overview of an Authorization Code Interception Attack.

Message 1: The application running at the end device makes an authorization request to the authorization server via the operating system/browser. The message sent from the application to the operating system/browser is sent through a safe API that cannot be intercepted (may potentially be intercepted in advanced attack scenarios). When the message is sent from the operating system/browser to the authorization server it is protected by TLS (OAuth requirement), which means that the message cannot be intercepted.

Message 2: The authorization server sends the authorization code to the operating system/browser.

Message 3: The authorization code is sent to the requester (the application) via the redirection endpoint URI that was given in the authorization request. However, it is possible for a malicious application to register itself as a handler to the legitimate application. This means that the malicious app can intercept the authorization code.

Message 4: The malicious application sends a token request to the authorization server by including the authorization code intercepted in the previous step.

Message 5: The authorization server sends a token response to the malicious application.

However, there are pre-conditions for this attack to happen:

- The operating systems must allow multiple application registrations, since the malicious application needs to be registered by the attacker on the client device. A custom URI scheme must be enabled to be registered by multiple applications.
- The OAuth 2.0 authorization code grant is used.
- The attacker has access to the "client_id" and the "client_secret", if provisioned.
- The attacker is either able to observe the response from the authorization endpoint at the authorization server via the installed application, or responses *and* requests. The attacker is not able to act as a man in the middle. The first scenario is mitigated when "code_challenge_method" value is "plain". The second (and first) scenario is mitigated when "code_challenge_method" value is "S256" or another value defined by a cryptographically secure "code_challenge_method" extension.

5.2 PKCE Protocol Flow

This section describes the PKCE protocol flow, that adds additional parameters to the OAuth 2.0 authorization code flow to mitigate the attack described in the previous section.

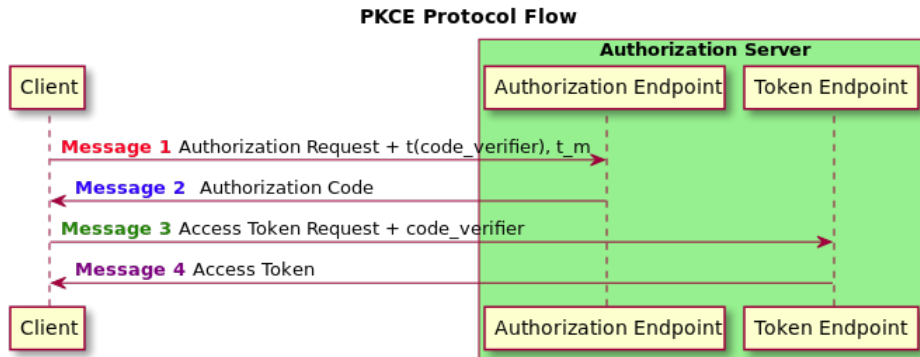


Figure 5.2: An overview of the PKCE Protocol Flow.

Message 1: The client creates a secret, called "code_verifier", that it records. See Section 5.2.1 for more details. The "code_verifier" is derived into a transformed version "t(code_verifier)" that is called the "code_challenge", i.e., "t(code_verifier)"="code_challenge". See Section 5.2.2 for more details. The "t(code_verifier)" is sent to the authorization server at the authorization endpoint together with the authorization request and the transformation method "t_m". See Section 5.2.3 for more details.

Message 2: The authorization endpoint records the "t(code_verifier)" and the transformation method "t_m". The authorization endpoint responds to the client with an authorization code, see Section 5.2.4 for more details.

Message 3: When the client sends the access token request, it includes the authorization code (as in the OAuth 2.0 protocol) and the "code_verifier" that it stored before sending Message 1. See Section 5.2.5 for more details.

Message 4: The token endpoint at the authorization server transforms the "code_verifier" and compares it to the "t(code_verifier)". See Section 5.2.6 for more details. If they are equal, it sends a successful token response. Otherwise, it sends a token error response.

5.2.1 Client Creates a Code Verifier

The code verifier is a high-entropy cryptographic random string, that has a length between 43-128 characters. The code verifier should have enough entropy to make it hard to guess, and it is recommended to create a 32-octet sequence (by using a suitable random number generator). The octet sequence is then base64url-encoded to produce a 43-octet URL safe string to use as the code verifier. The following is the ABNF (Augmented Backus-Naur Form) for the code verifier:

```

code-verifier = 43*128 unreserved
unreserved = ALPHA / DIGIT / "-" / "." / "_" / "~"
ALPHA = %x41-5A / %x61-7A
  
```

DIGIT = %x30–39

5.2.2 Client Creates the Code Challenge

The code verifier is transformed into a code challenge by using the "plain" transformation, or the "S256" transformation:

```
plain
    code_challenge = code_verifier
S256
    code_challenge = BASE64URL-ENCODE(SHA256(ASCII(
        code_verifier)))
```

The client must use the "S256" transformation if is capable of doing so, since it is Mandatory To Implement (MTI) on the server.

The following is the ABNF for the code challenge:

```
code-challenge = 43*128 unreserved
unreserved = ALPHA / DIGIT / "-" / "." / "_" / "~"
ALPHA = %x41–5A / %x61–7A
DIGIT = %x30–39
```

5.2.3 Client Sends the Code Challenge with the Authorization Request, Message 1

The parameters for the authorization request are defined in the OAuth 2.0 protocol, see Section A.1.1, except for the following additional parameters defined in this protocol:

```
//REQUIRED PARAMETERS:
code_challenge
    The code challenge
//OPTIONAL PARAMETERS:
code_challenge_method
    The values are "S256" or "plain", the default value is
    "plain" if the parameter is not included
```

5.2.4 Server Returns the Code, Message 2

The server must associate the values of the "code_challenge" and the "code_challenge_method" with the authorization code before sending the authorization response, either by storing the values encrypted in the code or store them on the server associated with the code. However, the server must not include the value of the "code_challenge" in the client requests that in a form that other entities can extract.

Error Response

The authorization endpoint at the authorization server must return an authorization error response with the "error" value set to "invalid_request", if the "code_challenge" parameter is not included in the request. The "error_description" (or the response of the "error_uri") should be, for example, code challenge required. The authorization endpoint at the authorization server must return an authorization error response with the "error" value set to "invalid_request", if the requested transformation is not supported by the server. The "error_description" (or the response of the "error_uri") should be, for example, transform algorithm not supported.

5.2.5 Client Sends the Authorization Code and the Code Verifier to the Token Endpoint, Message 3

The parameters for the token request are defined in the OAuth 2.0 protocol, see Section A.1.4, except for the following additional parameters defined in this protocol:

```
//REQUIRED PARAMETERS:  
code_verifier  
    The code verifier
```

5.2.6 Server Verifies code_verifier before Returning the Tokens

The token endpoint at the authorization server must verify the "code_verifier" by using the "code_challenge_method" that was bound to the authorization code before the authorization code was sent in Message 2. The token endpoint at the authorization server does the following comparison to verify the code_verifier:

```
If the "code_challenge_method" was "S256":  
    BASE64URL-ENCODE(SHA256(ASCII(code_verifier))) ==  
        code_challenge  
If the "code_challenge_method" was "plain":  
    code_verifier == code_challenge
```

If the values are equal, the token request will be validated according to the OAuth 2.0 protocol. Else, an error response with the "error" value set to "invalid_grant" must be returned.

This chapter describes some of the security threats related to the processes of authentication and authorization. The reason why these specific security threats are presented, is because they are mentioned in other parts of the report.

6.1 Cross-Site Request Forgery

Cross-Site Request Forgery (CSRF) is a malicious exploit of a website where an attacker forces the victim user's web browser to perform unauthorized commands on a trusted website without the victim user's interaction in that action. CSRF is also known as one-click attack or session riding. CSRF is sometimes incorrectly confused with XSS (Cross site scripting). However, since they are not the same and protection against XSS does not protect against CSRF. In XSS, the user trusts the website integrity, and gets tricked to give direct information to the attacker. In CSRF, the website trusts the user's requests and accomplishes any kind of action that comes from his flagged authentication, which indirectly gives advantages to the attacker.

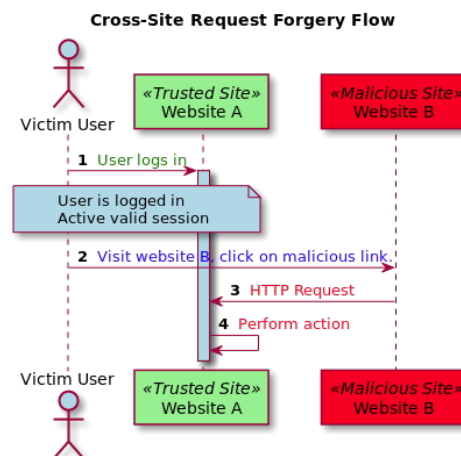


Figure 6.1: An overview of a Cross-Site Request Forgery Attack.

Figure 6.1 provides an overview of a Cross-Site Request Forgery that includes a victim user, a trusted site (Website A), and a malicious site (Website B).

1: The user logs in on a trusted website, website A. This creates an active valid session with the trusted website.

2: While logged in to website A (working in a valid session), the user visits a website, website B. On website B, an attacker has already posted a malicious link that sends a HTTP request to website A when the link is clicked. This HTTP request requires a valid session of the user to perform some valid action on website A, which is the case in this example. The user clicks the link on website B, by mistake.

3: The HTTP Request is sent to website A.

4: The HTTP Request uses the valid session of the user to perform some valid action on website A.

The following is an example of a malicious link using an image tag:

```

```

There are more than one way of sending a malicious link to a victim user, e.g., by a third party website as explained above, by email, and on instant messaging. Request forgeries require prior knowledge but is simple to exploit, and the most popular ways to execute CSRF attacks is by using a HTML image tag or a JavaScript image object.

6.1.1 Protection Against CSRF

There are some guidelines that help preventing a CSRF attack [4]:

1. Use *random tokens* that are unpredictable. Random tokens should be generated for each form submission and each authentication process. This means that it's difficult for the attacker to guess the next random pattern to fill in malicious URL.
2. Use *post for form submission* rather than get. When using the get method for form submission, the variables and values in the URL can be seen by anyone.
3. *Limit the life-time of authentication cookies*. If the user visits other websites, the cookies will expire after a short period of time and the valid session will expire.
4. Include *damage limitation* on websites to reduce the damage of CSRF. If an attacker manages to perform a CSRF attack, actions performed by the attack can be prevented if they require authentication every time.
5. *Force user to use the form of the (trusted) website*.

An example of a protection mechanism against CSRF is defined in the OpenID Connect protocol. When making authentication requests defined in the OpenID Connect protocol, see (for example) Section B.1.4, a request parameter "state" is included. This is an opaque value that is used to maintain the state between the request and the callback. The value of the "state" parameter is usually cryptographically bound with a browser cookie, with the purpose of mitigating CSRF. [16]

6.2 Brute Force Attacks

A brute force attack, or a "password guessing attack", is an attack that discovers a password by trying every possible combination of numbers and letters. A user showing carelessness in choosing username and password, i.e., choosing a simple password that might be based on user information, is vulnerable to a brute force attack. There are tools available today that can be used for brute force attacks that creates different password combinations. The tools use different IP addresses on each try, which makes it more difficult to trace a single account for unsuccessful password attacks.

6.2.1 Protection Against Brute Force Attacks

To prevent brute force attacks, passwords for user account should follow a certain password policy, e.g., minimum password length, contain both upper and lower-case letters, numeric characters, and special characters.

A common technique to avoid brute force attacks is to lock an account for a certain period of time (e.g., one hour or one day) after a certain number of incorrect password attempts. Another way to avoid brute force attacks is to use CAPTCHA (Completed Automated Public Turing test to tell Computers and Humans Apart). This technique can help avoid brute force attacks, since machines might have issues to enter the CAPTCHA codes [5]. However, modern advanced technologies such as deep learning is capable of recognising captcha without human intervention [6].

In the resource owner password credentials flow, see Section 3.7, the authorization server must protect the resource owner's password against brute force attacks, since the access token request from the client includes the resource owner's password. This can be done by, for example, using rate-limitation or generating alerts. [7]

6.3 Replay Attack

One definition of a replay attack is: "an attack on a security protocol using a replay of messages from a different context into the intended (or original and expected) context, thereby fooling the honest participant(s) into thinking they have successfully completed the protocol run." [37].

6.3.1 An example of a protection mechanism against replay attacks

An example of a protection mechanism against replay attacks is defined in the OpenID Connect protocol. When making authentication requests defined in the OpenID Connect protocol, see (for example) Section B.1.4, a request parameter "nonce" is included. This is a string value that is used to associate a client session with an ID token and to mitigate replay attacks by having sufficient entropy of the value. [16] E.g., a web server client can store a cryptographically random value as a HttpOnly session cookie and use a cryptographic hash of the value as the "nonce" parameter. The nonce in the returned ID token can then be compared to the hash of the session cookie to detect ID token replay by third parties. [17]

6.4 Clickjacking

A clickjack attack is an attack on an Internet-based application, a social media system, or a browser, which tricks the user into accessing malware [18]. Clickjacking can be more harmless than other malicious threats, since many of the attacks are intended to trick users into clicking on a picture or a link that will redirect the user to sites that pay the perpetrator for page views. However, there are clickjacking attacks that can use malicious scripts to deliver malware and junk mail, to phish for account information or to trick users into signing up for unwanted services. [19]

An example of a clickjacking attack, described by the OAuth 2.0 protocol [3], is when an end-user (resource owner) clicks a misleading visible button on a malicious site that grants the attacker's client access to a specific resource without the end-user's knowledge. This attack can happen if the attacker registers a legitimate client and constructs a malicious site, and the malicious site loads the authorization server's authorization endpoint web page in a transparent iframe overlaid on top of a set of dummy buttons (constructed to be placed under important buttons on the authorization page).

This chapter describes the results of this project. Section 7.1 gives the results and overview of how the authentication process is a part of the OAuth 2.0 Authorization Framework, OpenID Connect and Proof Key for Code Exchange. The process, theory and background of the authentication process of the multi-factor authenticator that is the result of this project is described in Section 7.2. The implementation of the multi-factor authenticator is described in Section 7.3.

7.1 The Authentication and Authorization Process

There are several problems and limitations in the traditional client-server authentication model. The client requests a protected resource on the server, using the resource owner's credentials for authentication. The resource owner's credentials are shared with third-party application, in order to provide the third-party application to access restricted resources. This creates several problems, for example:

- the resource owner cannot revoke the access token to one third-party application without revoking access to all third parties (by changing password),
- the third-party applications are required to store the resource owner's credentials (password in clear-text),
- compromise of any of the third-party applications means compromise of the resource owner's credentials (the resource owner's data is no longer protected),
- and despite the security weaknesses inherent in passwords, servers are required to support password authentication.

OAuth provides a method for users to grant third-party access to their resource *without sharing their passwords*, as well as granting limited access to the resources, e.g., limited scope (which resources the third-party can access), duration (how long the third-party can access the resources), etc. In OAuth, the client requests access to the resource that is controlled by the resource owner and hosted by the resource server. The client is issued a different set of credentials than those of the resource owner. Third-party clients are issued access tokens, with a specific scope, lifetime, and other attributes, that is used to access the protected resources hosted by the resource server.

OpenID Connect provides extensions to the OAuth 2.0 authorization process by enabling clients to verify the identity of the end-user based on the authentication performed by an authorization server, it also enables clients to obtain basic profile information about the end-user in an interoperable and REST-like manner. To use the OpenID Connect as an extension to OAuth 2.0 the "openid" scope value is included in the authorization request. The information about the authentication is returned in an ID Token that is a JSON Web Token (JWT). In the process of authorization and authentication process of the OpenID Connect, the following main roles are:

- **End-User/Resource Owner:** The resource owner is an entity that is capable of granting access to a protected user. It is referred to as an end-user when the resource owner is a person.
- **User-Agent:** Typically a web browser that the end-user is interacting with.
- **OIDC Relying Party:** The OpenID Connect Relying Party (RP) is an OAuth 2.0 client application that requires end-user authentication and claims from an OpenID provider.
- **OIDC OpenID Provider:** The OpenID Provider (OP) is an OAuth 2.0 authorization server that is capable of authenticating the end-user, as well as providing claims to a relying party about the authentication process and the end-user. The OP usually consists of an authorization endpoint, a token endpoint and a userinfo endpoint.

As explained in Chapter 3, the OAuth 2.0 protocol defines four different flows: authorization code, implicit, resource owner password credentials, client credentials, as well as an extension mechanism for defining additional grant types. The OpenID Connect provides its extended functionality to the following flows: authorization code, implicit, and hybrid flow (a mix between the authorization code flow and the implicit flow). Since the multi-factor authenticator that is a result of this project is a part of the flow defined in the OpenID Connect protocol, all other flows than the authorization code flow, the implicit flow, and the hybrid flow, are ignored. The different flows determines how the ID token and access token are returned to the client. The main difference between the authorization code flow and the implicit flow is that the application receives the access token (and optionally the ID token) directly from the server in the implicit flow, while the application receives an authorization code in the authorization code flow. The implicit flow should be used when the application (typically a JavaScript app) running within a browser is not trusted with a client secret. The used client and login-application running on Axis's server in this project is considered to be trusted, which means that the authorization code flow is used. The following picture shows the overview of the authorization code flow and where the authentication by the multi-factor authenticator will be executed:

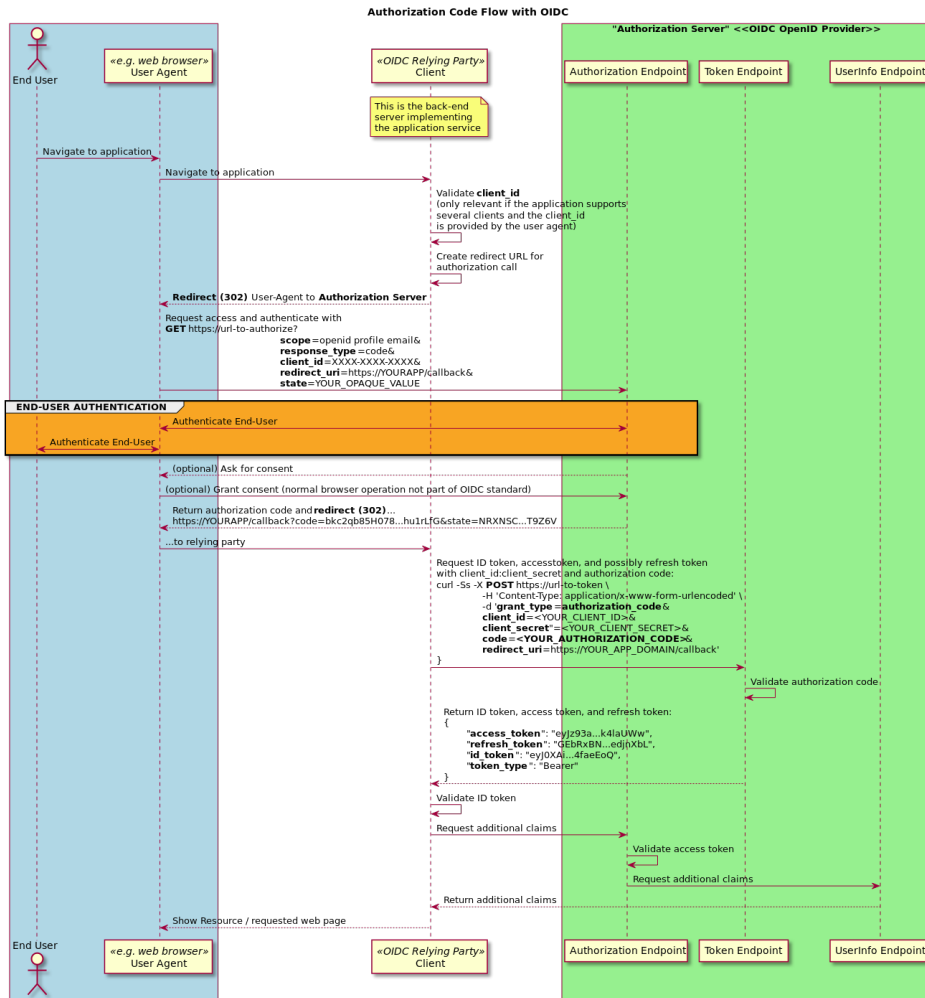


Figure 7.1: An overview of the authorization code flow and where the authentication by the multi-factor authenticator will be executed.

More details about the authorization code flow can be found in Section 4.7.

The authorization code flow is optimized for confidential clients, i.e., clients that are able to keep a client secret. If the authorization code flow is used with public clients they are susceptible to the authorization code interception attack, see Section 5.1. This attack is mitigated by dynamically create a cryptographically random key called "code verifier". The PKCE specification adds additional parameters to some requests and responses in the authorization code flow, to prevent an attacker from intercepting the authorization code in the authorization code flow. The authorization code interception attack and the Proof Key for Code Exchange is described more in Chapter 5.

7.2 Multi-Factor Authenticator

The performance of a multi-factor authenticator is determined by the set of which authentication factors that are used, see Chapter 2. The performance can be measured in *security*, *usability*, and *cost*. The following factors were used in this authenticator: knowledge factor, ownership factor, and biometric factor. This multi-factor authenticator has the following higher-level authentication processes for the user:

1. User identifies and verifies its identity with a traditional username/password authenticator.
2. User opens the fingerprint application installed on its smartphone.
3. User authenticates by using the fingerprint sensor on the smartphone.
4. User is authenticated.

After the user is authenticated, an authorization code will be sent from the authorization endpoint at the authorization server (OIDC OpenID Provider) to the client (OIDC Relying Party), as shown in picture 7.1.

The multi-factor authenticator that is the result of this project describes solutions to some of the challenges that are found and presented in this project for multi-factor authenticators (that also involve the biometric authentication factor). This is described in Section 7.2.4.

7.2.1 First step of Authentication, HTML login

The first step of authentication is the traditional username/password authenticator, where the user identifies with its email and verifies with its password in a HTML login form. This authenticator was already implemented in the current Axis-login system. Therefore, technical implementation details were not investigated. However, studies have shown that the traditional username/password authenticator is the weakest level of authentication, compared to authenticators that includes other factors than the knowledge factor. The password should also have minimum requirements of length and complexity. Since the username/password authenticator is considered to be low in security but high in usability, it is appropriate to have it as a first-step authenticator. The user only needs to remember a password or a passphrase to authenticate in this step. This is only one step of the authentication process of the multi-factor authenticator, which means that the overall security level of the authenticator is higher than the level of security of the first-step authenticator. The second step of the authentication process of the multi-factor authenticator is described in Section 7.2.2 and 7.2.3, if a device is not registered or if a device is registered, respectively.

7.2.2 Second step of Authentication during Registration, Device registration

When the user has authenticated using the username/password authenticator, the user ID of the authenticated user is fetched. If no devices are registered for the

user, the user will be redirected to a page where the user can choose to register a new device. If the user chooses to do so (which is needed for the user to be authenticated with this authenticator), the user can click a link that redirects the user to a page where a QR-code is displayed. The QR-code contains four parameters:

- A link to the fingerprint application that will trigger a smartphone to open the fingerprint application when the QR-code is scanned
- A link to the authentication endpoint
- A link to the registration endpoint
- A shared Time-Based One Time Password (TOTP) secret

The TOTP secret is a long randomized string. The register page with the QR-code waits for a HTTP POST request from the registration endpoint. The HTTP request from the registration endpoint will be sent to the register page after it has received a HTTP GET request from the fingerprint application.

The user opens the smartphone's camera, or any other QR-scanner application, scans the QR code and gets a pop-up notification to open the fingerprint application. When the fingerprint application opens, it saves the authentication endpoint, the registration endpoint, and the TOTP secret in the background. The user is asked to authenticate by placing its finger on the fingerprint sensor. The fingerprint authenticator application will trigger an authentication success if the fingerprint is the same as the fingerprint that is registered for the user on the device in advance to the authentication. If there is a new registration endpoint, authentication endpoint, and TOTP secret registered in the fingerprint application, the fingerprint application will generate a TOTP and include it in a HTTP GET request to the registration endpoint (if there are no new endpoints or secrets, the application will send the TOTP to the authentication endpoint).

The TOTP is generated by concatenating the TOTP secret with the current interval (default value is 30 seconds), and hashing and truncating the concatenated string. The hash is created by using a cryptographic hash function (default algorithm is Secure Hash Algorithm 1, SHA-1), and the resulting TOTP is a 6 digits long code.

After the HTTP GET request is received at the registration endpoint, the TOTP will be validated. If the TOTP is valid, a HTTP POST request is sent to the registration page, and a randomized device ID will also be created and added for the user. The device ID will be saved and connected to the secret, the TOTP used for registration, and the TOTP timestamp in the database.

The default value for a TOTP in this authenticator is 30 seconds, i.e., the user has 30 seconds to scan the QR code, open the app and press its finger on the fingerprint sensor. The TOTP will not be valid if the timestamp has exceeded 30 seconds. The user can choose to restart the registration process, which will generate a new QR-code with a new randomized TOTP secret.

7.2.3 Second step of Authentication, Device already registered

When the user has authenticated using the username/password authenticator, the user ID of the authenticated user is fetched. If there is a device registered for the user, the authentication page waits for a HTTP POST request from the authentication endpoint. The user then opens the fingerprint application on its smartphone. When the fingerprint application opens, it already has the authentication endpoint, the registration endpoint, and the TOTP secret saved from the registration described in Section 7.2.2. The user is asked to authenticate by placing its finger on the by placing its finger on the fingerprint sensor. The fingerprint authenticator application will trigger an authentication success if the fingerprint is the same as the fingerprint that is registered for the user on the device in advance to the authentication. The fingerprint application will generate a TOTP and include it in a HTTP GET request to the authentication endpoint. The TOTP is generated by concatenating the TOTP secret with the current interval (default value is 30 seconds), and hashing and truncating the concatenated string. The hash is created by using a cryptographic hash function (default algorithm is Secure Hash Algorithm 1, SHA-1), and the resulting TOTP is a 6 digits long code. After the HTTP GET request is received at the authentication endpoint, the TOTP will be validated. If the TOTP is valid, a HTTP POST request is sent to the authentication page, and the device ID will be connected to the used TOTP, the used TOTP secret, and the TOTP timestamp in the database. Since the default value for a TOTP in this authenticator is 30 seconds, the user has 30 seconds to open the app and press its finger on the fingerprint sensor. The TOTP will not be valid if the timestamp has exceeded 30 seconds. The user can choose to restart the authentication process, which will generate a new TOTP and update the timestamp of when the authentication process started.

7.2.4 Solutions to challenges of Multi-Factor Authenticators that includes Biometric Factors

Chapter 2 describes and compares commonly used authenticators based on different authentication factors. There are several challenges or security threats that are described in the chapter that the resulting multi-factor authenticator of this project mitigates.

The correlation between the user identity and the identity of the smart sensor on the device (e.g., fingerprint sensor or application) must be established for security reasons. It is important that only the authenticated user has access rights to the application or sensor. The solution to this challenge is to generate a random device ID that is connected to the user ID during a successful registration, see Section 7.2.2. It is only possible to fetch the user ID if there is an authenticated user, i.e., if the user was successfully authenticated by the username/password authenticator. The fingerprint application is only accessible if the user has a screen lock on the smart phone, i.e., if the user has a PIN code, a fingerprint, iris recognition or face recognition registered for screen lock. Otherwise, the fingerprint application will give the user a warning and the user will not be able to authenticate using the application. Moreover, the fingerprint application is not possible to use if there are

no fingerprints registered for the user on the device, i.e., the user has to register a fingerprint on the device in advance to using the fingerprint application.

The disadvantage of only allowing an application to fully function if the user has enabled screen lock might decrease the usability, since it is another step for the user. On the other hand, if the user has, for example, fingerprint verification as a screen lock the user only has to press a button to unlock the screen.

The password has to be stored and connected to the user ID, but the password should never be stored in plain text. The best practices of storing a password includes generating a random string salt that is concatenated with the password to protect against dictionary attacks. The generated string should then be hashed with a cryptographic function and stored instead of the plain text password. In this multi-factor authenticator, no password is stored in plain text, except for the TOTP secret. However, the TOTP secret is not directly connected to the user ID. The user ID is connected to the device ID, and the device ID is connected to the TOTP secret. The TOTP secret is a random 30 byte string, and there is a security vulnerability with storing it in case an attacker would be able to steal it. On the other hand, the TOTP that is used for authentication is generated by hashing and truncating the value of a combination of the secret ("salt") with the current time-interval.

The danger of eavesdropping is eliminated when OTPs are used because once an OTP is used, it is no longer useful. The motivation for using OTPs is that if one OTP is compromised, it will not affect the security of sessions involving another OTP. Furthermore, breaking a one-time password system that is implemented properly, requires sophisticated, active attacks that are beyond the abilities of most attackers [42]. Attackers can steal codes through targeted phishing attacks. This is notably the case for the SMS authenticator. A way to mitigate this type of threats is to invalidate a code after a short time, and limit the number of failed attempts to log in with a code. This is one of the reasons why TOTPs are used in this multi-factor authenticator instead of OTPs. The TOTP is only valid for a short time period, regardless of whether the TOTP has been used or not. The current time interval in the implementation is 30 seconds, which is less user-friendly than a longer time-interval, but it brings a higher level of security.

A security threat of biometric authentication is that an attacker can intercept the information and replay it, since the information that the user provides is generally similar at each authentication attempt. The solution to this in this implementation is that the fingerprint never leaves the device. A fingerprint has features of fingerprint ridges [36]. The TOTP that is generated in the application has no relation to the fingerprint itself in terms of the features of the fingerprint. The already built-in fingerprint sensor and manager in the smart phone determines if the fingerprint is valid or not. If the fingerprint would leave the device, it could be exposed to attacks and, if compromised, no longer safe to use for any authentication process.

A study, see Section 2.1.3, showed that users felt fingerprint unlock a little or a lot more secure than a PIN, but this is not true due to PIN being a fallback mechanism on most smartphones. The same applies for authenticators. The fallback mechanism, i.e., the case when the user loses its device and have to use another authenticator, has to be of the same level of security as the original authenticator. This means that the multi-factor authenticator that is the result of this project *must* have a fallback authenticator that has the same level of security. If this same level of security is defined to be using the same authentication factors, it must have a fallback authenticator that includes biometric authentication. If the user loses its device with the fingerprint authenticator, there must be another device that the user can use to authenticate with biometrics (fingerprint, iris recognition, face recognition, etc.) that has to be registered before this scenario occurs (reasonably during registration of the first device). A study of account recovery of 10 popular web services was made during this project. This study showed that many web services did not have a fallback authenticator that was of the same level of security as the original authenticator. Hence, using biometric authenticators can give users incorrect trust for some services.

Due to the importance that only the authenticated user has access rights to the application or sensor, a biometric authenticator should not be used as a standalone authenticator. As mentioned above, the correlation between the user identity and the identity of the smart sensor must be established for security reasons. To confirm the correlation between the two identities and to give the user access rights to the application or sensor, the user needs to be authenticated with some authentication factor before using the hardware device with the biometric sensor. The solution to this in the resulting authenticator is to have the traditional username/-password as a prerequisite to the fingerprint authenticator.

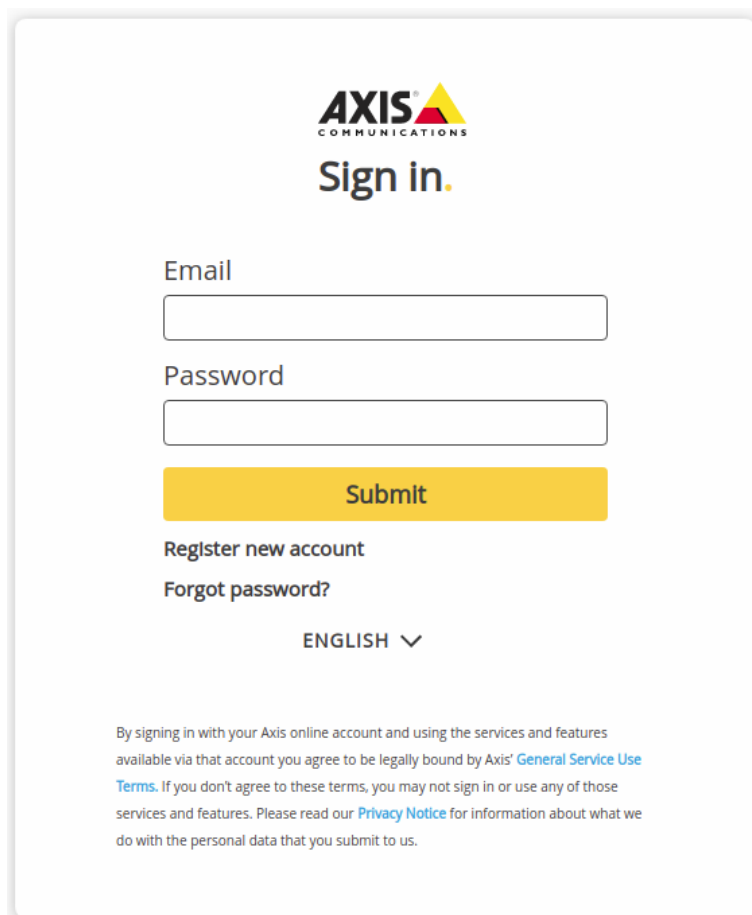
NIST policy on hash functions [40], states that the use of the cryptographic hash function SHA-1 may still be used in the context described above. However, the use of SHA-2 or SHA-3 is considered more secure against some security attacks [41].

7.3 Implementation

The implementation results of a multi-factor authenticator into the currently used login-application consists of a plugin and an application for Android devices. The plugin implements the necessary functionality that is required for implementing biometric authentication, via an Android device, into the currently used system. The Android application uses the fingerprint sensor on the device to authenticate the user. When the user is authenticated a time-based one time password (TOTP) is generated and sent to one of the system's endpoints (the registration endpoint during device registration or the authentication endpoint). The plugin implementation is described in Section 7.3.1, and the implementation of the android application is described in Section 7.3.2.

7.3.1 Plugin implementation

The functionality of the current system can be extended through plugins. The fingerprint authenticator plugin is a plugin JAR that contains implementations of the current system's extension points, as well as other library and dependency JARs. The plugin JAR and the dependency JARs are installed on the server after being dropped in a specific directory. The plugin is implemented in Java and the development, deployment, and execution framework used for the plugin is the Maven framework. The plugin contains all the necessary back-end functionality and front-end implementations for the user interface. The plugin implements two additional endpoints, a registration-endpoint and an authentication-endpoint. The HTML-login, that is already implemented in the current system, is chosen as a prerequisite to the fingerprint authenticator plugin. The front-end part of the HTML-login is shown in Figure 7.2.



AXIS
COMMUNICATIONS

Sign in.

Email

Password

Submit

[Register new account](#)

[Forgot password?](#)

ENGLISH ▾

By signing in with your Axis online account and using the services and features available via that account you agree to be legally bound by Axis' [General Service Use Terms](#). If you don't agree to these terms, you may not sign in or use any of those services and features. Please read our [Privacy Notice](#) for information about what we do with the personal data that you submit to us.

Figure 7.2: The page of the HTML-login.

After this step, the plugin checks if the user is authenticated. A requirement

for the fingerprint plugin to work is that the user is authenticated by another authenticator (username/password in this case). The user ID and the registered devices for the user ID are fetched from the account manager. If a device is not registered for the user, the front-end page shown in Figure 7.3 is displayed to the user.

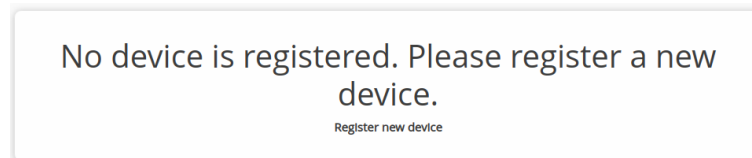


Figure 7.3: The page that is displayed to the user if there is no device registered for the user.

If the user clicks the "Register new device" button a QR-code will be created including the registration-endpoint, the authentication-endpoint and the TOTP secret. The TOTP secret is a 20-bytes long randomized code. The registration page, shown in Figure 7.4, waits for a HTTP POST request from the registration-endpoint.

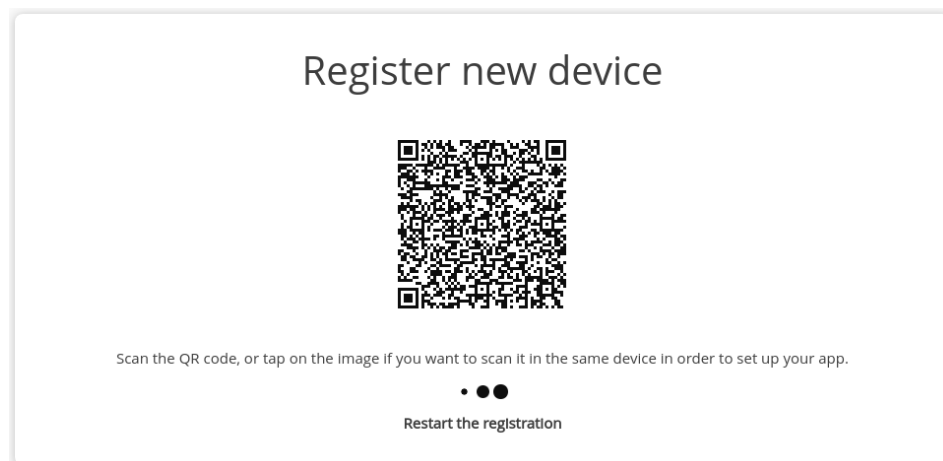


Figure 7.4: The registration page.

If the user clicks the "Restart the registration" button, a new QR-code will be generated containing a different TOTP secret.

At the same time as the TOTP secret is generated, a nonce token issuer is issued with the TOTP secret as a reference. The reason why two TOTP secrets are used as a reference is because there might be a delay window. A delay window is a time difference that might occur if the current interval is calculated with just a small time difference. This will create a different TOTP. The current implementation has one delay window. Therefore, two different TOTP secrets

are accepted. When the registration endpoint gets a HTTP GET request containing a TOTP it is validated as a nonce token. If the TOTP is registered as a nonce and a device ID belonging to the current user ID has a correlation to the TOTP, the timestamp of the TOTP will be checked. If the timestamp is within the current interval (30 seconds), the user is authenticated. If the timestamp is not within the current interval, the user will have to restart the registration process.

The next time the user authenticates with the multi-factor authenticator a device is registered for the user, and the authentication page shown in Figure 7.5 is displayed to the user.

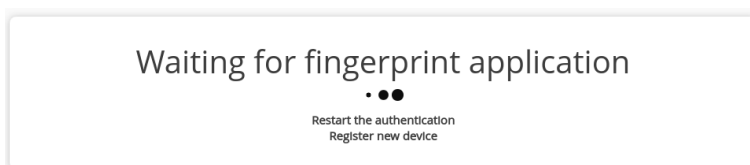


Figure 7.5: The authentication page.

When the authentication page is shown to the user the TOTP secret are saved as a reference, in the same way as for the registration process. The TOTP that is sent to the authentication endpoint is also validated in the same way as for the registration process. If the TOTP is valid, the user is authenticated. The user can choose to "Restart the authentication", new TOTP secret will then be saved as references. The user can also choose to "Register new device", which will trigger the registration process to start and if a new device is registered it will replace the old one. If the same device is registered more than once, the new registration will replace the old one.

7.3.2 Android Application

When the Android application is triggered by a QR-code, the parameters registration-endpoint, authentication-endpoint and the TOTP secret will be saved in the application. At the same time as these parameters are saved, the application will send the generated TOTP to the registration-endpoint upon authentication success. For the authentication process to start, there are a few conditions that are checked:

1. The device needs to have a fingerprint sensor
2. The device must have a fingerprint registered for the user
3. The lock-screen needs to be protected with a PIN, a password, or any biometric factor that is possible to register on the device

If the fingerprint is incorrect, the authentication fails and the app view in Figure 7.6 is displayed to the user.

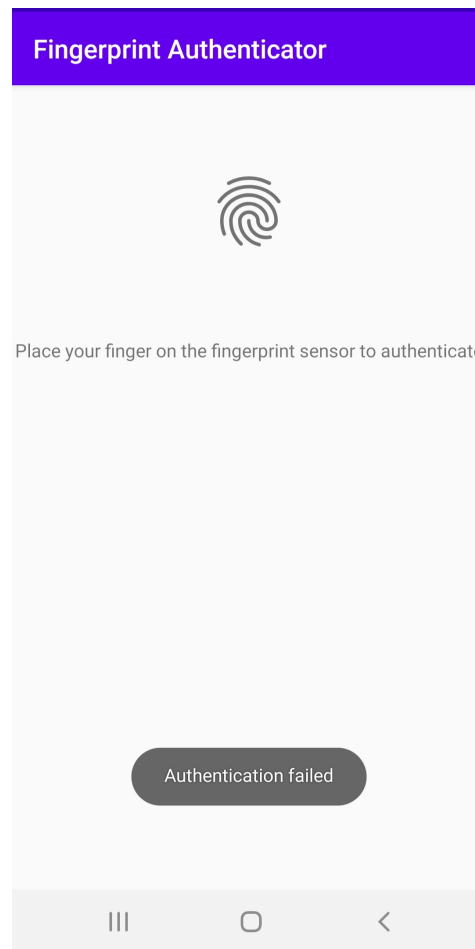


Figure 7.6: App view, Authentication Failed

If the fingerprint cannot be read, due to the finger not covering the whole sensor or if the user does not hold the finger on the fingerprint sensor long enough, there is an authentication help that is displayed to the user, see Figure 7.7.

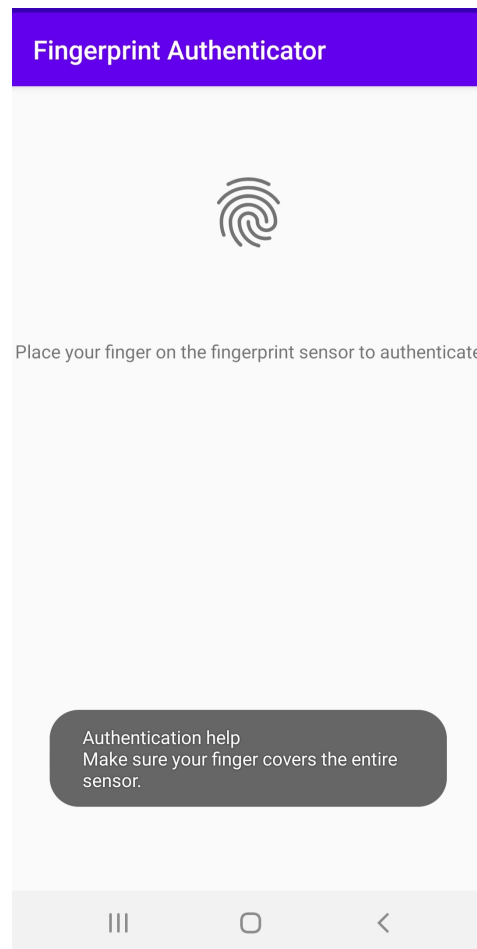


Figure 7.7: App view, Authentication Help

If there is an authentication error, for example, if the user closes the application in the middle of an authentication process, an authentication error message will be displayed.

On authentication success, a TOTP is created from the saved TOTP secret and sent with a HTTP GET request either to the registration-endpoint or the authentication-endpoint. The app view that is displayed to the user on authentication success can be seen in Figure 7.8.

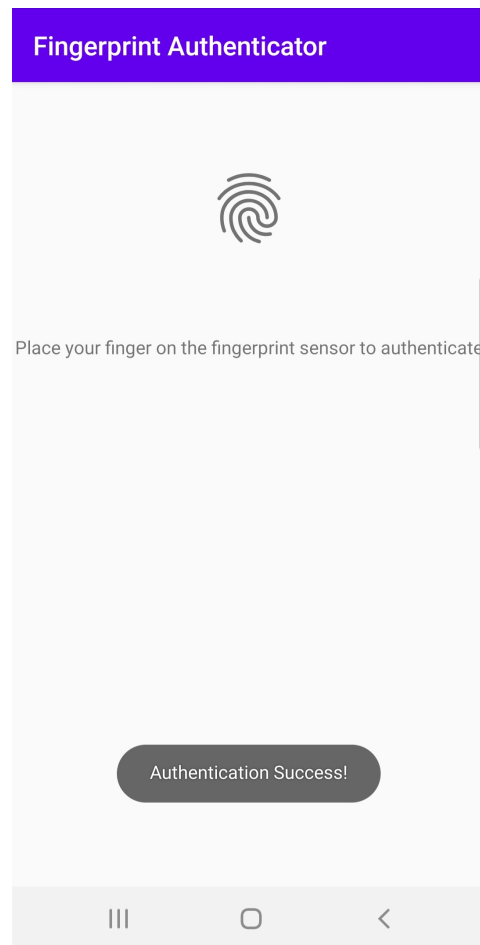


Figure 7.8: App view, Authentication Success

Conclusion

The specified goals of this project were to gain in-depth knowledge of the OpenID Connect Protocol and Multi-Factor Authentication in the efforts to implement an improved solution of a multi-factor authenticator, do research about authentication methods and technologies based on authentication factors, and implement a multi-factor authenticator into the currently user login-application.

In summary, all goals of this projects were achieved. However, the processes of authentication and multi-factor authentication is constantly developing and improved. The OpenID Connect protocol is a widely used protocol and is an identity layer on top of the OAuth 2.0 protocol. To be able to gain in-depth knowledge about the OpenID Connect protocol, it was also required to gain in-depth knowledge about the OAuth 2.0 protocol. The core functionality of OpenID Connect is to add authentication to OAuth 2.0 and use Claims to communicate information about the end-user. There are also security and privacy considerations covered by the protocol.

By implementing a multi-factor authenticator into the currently used login-application, the security risks, usability concerns and challenges with user authentication were easier to pinpoint. One main challenge with all authenticators, that relates to security, is *what* to save and *how* to save it. Passwords, PINs, security tokens, one-time passwords, or biometric data. The most important result that was found during the research on commonly used authenticators based on authentication factors together with a study on popular web services, was that the overall level of security of the authenticator is the minimum of the level of security of the factor normally in use and to the level of security of its fallback method. The fallback method is used in case the user loses, for example, its email address or device. If the authenticator has a biometric factor, the fallback authenticator *must* have a biometric factor to maintain the level of security that the biometric authenticator brings. Another result is the importance of usability of an authenticator. If an authenticator is too complicated for a user to use, in terms of authentication time or user errors, it will not be a good authentication method. Furthermore, if the user does not understand the importance or level of security (low or high) that one authenticator brings, the user will be the weakest component of the authentication system. There is always a trade-off in usability and security, but fast solutions like pressing a finger on a fingerprint sensor brings both usability and security, it might

even be considered to be easier than entering a password.

The implemented multi-factor authenticator implements biometric authentication in a way that does not involve biometric data leaving a local device or being stored on a server, since it combines an automated process of time-based one time passwords with biometric authentication. The reason for our approach was the security vulnerability and the privacy aspect in biometric factors being stored on a server or leaving a local device. A user's biometric data cannot be changed if compromised, compared to a password, and biometric data can identify a user for their entire lifetime. The use of biometrics poses privacy concerns since biometric data may reveal sensitive information about a person, facilitate discrimination, profiling, and mass surveillance. The leakage of biometric data can lead to privacy concerns that lasts a lifetime. Biometric factors are unique for every person, but if the factors get stolen or compromised, they might no longer be safe to use in authentication processes. Moreover, the time-based one time password that is created on the local device, via biometric authentication, is more secure than, for example, a one time password, due to its time-limit. Due to the importance that only the authenticated user has access rights to the application or sensor, a conclusion was to *not* use biometric authentication as a stand-alone authenticator. The correlation between the user identity and the identity of the smart sensor or application can be confirmed if the user is authenticated, and this is why the first step of authentication in the implemented multi-factor authenticator is the traditional and user-friendly username/password authentication. In other words, the implemented multi-factor is based on the knowledge factor (password), the ownership factor (smartphone), and the biometric factor (fingerprint).

8.1 Improvements and Changes

There are some improvements or changes that can be considered in connection to the current implementation of the multi-factor authenticator.

The TOTP's that are generated in the fingerprint application are currently only 6 digits long. Since this is an automated process, i.e., the user does not have to enter the TOTP anywhere, it is better that the TOTP's that are sent to the registration- and authentication-endpoints should be longer. Dictionary attacks are more complicated to execute if the codes that are sent are longer.

Multiple devices for one user is currently not implemented. However, the user should be able to register more than one device if a fallback authentication method that includes another device is to be implemented. It does not require a lot of additional implementation effort, but any security issues with this should be investigated.

The user needs to complete the registration process, as well as the authentication process, within 30 seconds. This interval can be made longer to increase usability, but the trade-off of doing this is a decrease in security.

The implemented Android application should be changed to support other biometrics than fingerprint, e.g., face recognition or iris recognition. This will not require a lot of additional implementation effort.

An improvement that can be made to the current plugin implementation is sending push notifications to the fingerprint application. However, there are additional security vulnerabilities with doing this, as presented in Chapter 2, and these should be carefully investigated.

The fingerprint application for the Android device can also be implemented for other mobile operating systems, like iOS.

SHA-2 or SHA-3 is considered more secure against some security attacks compared to SHA-1. SHA-1 may still be used in the context of this application, according to NIST policy on hash functions [40]. However, enabling the use of SHA-2 and SHA-3 is a security improvement and will not require a lot of additional implementation effort.

Bibliography

- [1] IETF. 2020. Mission and principles. [<https://www.ietf.org/about/mission/>]. Accessed 2021-01-04.
- [2] IETF. 2020. RFCs. [<https://ietf.org/standards/rfcs/>]. Accessed 2021-01-04.
- [3] IETF Trust, Hardt, D. (Ed.). 2012. The OAuth 2.0 Authorization Framework. [<https://tools.ietf.org/html/rfc6749>]. Accessed 2021-01-04.
- [4] Mohd. Shadab, S. Deepanker, V. 2011. Cross site request forgery: A common web application weakness. [<https://ieeexplore-ieee-org.ludwig.lub.lu.se/stamp/stamp.jsp?tp=&arnumber=6014783>]. Accessed 2020-01-11.
- [5] Konark Truptiben, D. 2013. Brute-force Attack "Seeking but Distressing". [<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.359.8963&rep=rep1&type=pdf>]. Accessed 2021-01-11.
- [6] Gogineni, S. Suryanarayana, G. Swapna, N. 2020. Machine Learning Based Encoder-Decoder for Captcha Recognition. [<http://resolver.ebscohost.com.ludwig.lub.lu.se/openurl?sid=EBSC0:edsee&genre=chapter&issn=edsee.IEEEConferenc&isbn=9781728154619&volume=&issue=&date=20200901&page=222&pages=222-227&title=2020%20International%20Conference%20on%20Smart%20Electronics%20and%20Communication%20%28IC0SEC%29%2C%20Smart%20Electronics%20and%20Communication%20%28IC0SEC%29%2C%202020%20International%20Conference%20on&atitle=Machine%20Learning%20Based%20Encoder-Decoder%20for%20Captcha%20Recognition&bttitle=2020%20International%20Conference%20on%20Smart%20Electronics%20and%20Communication%20%28IC0SEC%29%2C%20Smart%20Electronics%20and%20Communication%20%28IC0SEC%29%2C%202020%20International%20Conference%20on&jtitle=2020%20International%20Conference%20on%20Smart%20Electronics%20and%20Communication%20%28IC0SEC%29%2C%20Smart%20Electronics%20and%20Communication%20%28IC0SEC%29%2C%202020%20International%20Conference%20on&series=&aulast=Gogineni%2C%20Saikiran&id=DOI:10.1109/IC0SEC49089.2020.9215439>]. Accessed 2021-01-11.

-
- [7] IETF Trust, Hardt, D. (Ed.). 2012. The OAuth 2.0 Authorization Framework. [<https://tools.ietf.org/html/rfc6749#section-4.3.2>]. Accessed 2021-01-11.
- [8] The OpenID Foundation. Sakimura, N. Bradley, J. Jones, Michael B. de Medeiros, B. Mortimore, C. 2014. OpenID Connect Core 1.0 incorporating errata set 1. [https://openid.net/specs/openid-connect-core-1_0.html]. Accessed 2021-01-14.
- [9] IETF Trust. Sakimura, N. Bradley, J. Agarwal, N. 2015. Proof Key for Code Exchange by OAuth Public Clients. [<https://tools.ietf.org/html/rfc7636>]. Accessed 2021-01-14.
- [10] ISO. 2013. ISO/IEC 29115:2013. [<https://www.iso.org/standard/45138.html>]. Accessed 2021-01-14.
- [11] [<https://tools.ietf.org/html/rfc6750>]. Accessed 2021-01-14.
- [12] IETF Trust. Jones, M. Hardt, D. 2012. The OAuth 2.0 Authorization Framework: Bearer Token Usage. [<https://tools.ietf.org/html/rfc6750#section-3>]. Accessed 2021-01-14.
- [13] IETF Trust. Jones, M. Campbell, B. Mortimore, C. 2013. JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants draft-ietf-oauth-jwt-bearer-05. [<https://tools.ietf.org/html/draft-ietf-oauth-jwt-bearer-12>]. Accessed 2021-01-15.
- [14] IETF Trust. Jones, M. Campbell, B. Mortimore, C. Goland, Y. 2014. Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants draft-ietf-oauth-assertions-18. [<https://tools.ietf.org/html/draft-ietf-oauth-assertions-18>]. Accessed 2021-01-15.
- [15] The OpenID Foundation. de Medeiros, B. (Ed.). Scurtescu, M. Tarjan, P. Jones, M. 2014. OAuth 2.0 Multiple Response Type Encoding Practices. [https://openid.net/specs/oauth-v2-multiple-response-types-1_0.html#Combinations]. Accessed 2021-01-18.
- [16] The OpenID Foundation. Sakimura, N. Bradley, J. Jones, Michael B. de Medeiros, B. Mortimore, C. 2014. OpenID Connect Core 1.0 incorporating errata set 1. [https://openid.net/specs/openid-connect-core-1_0.html#AuthRequest]. Accessed 2021-01-19.
- [17] The OpenID Foundation. Sakimura, N. Bradley, J. Jones, Michael B. de Medeiros, B. Mortimore, C. 2014. OpenID Connect Core 1.0 incorporating errata set 1. [https://openid.net/specs/openid-connect-core-1_0.html#NonceNotes]. Accessed 2021-01-19.
- [18] [<http://resolver.ebscohost.com/ludwig.lub.lu.se/openurl?sid=EBSCO%3aedsoro&genre=chapter&issn=&isbn=9780191884276&volume=&issue=&date=20191024&spage=&pages=&title=A+Dictionary+of+the+Internet&atitle=clickjack+attack&btitle=A+Dictionary+of+the+Internet&jtitle=A+Dictionary+of+the+Internet&series=&aulast=Ince%2c+Darrel&id=DOI%3a10.1093%2facref%2f9780191884276.013.4110&site=ftf-live>]. Accessed 2021-01-19.

- [19] Ince, D. 2019. A Dictionary of the Internet (4 ed.). Oxford University Press, published online. [<http://resolver.ebscohost.com.ludwig.lub.lu.se/openurl?sid=EBSCO:bwh&genre=article&issn=03624331&isbn=&volume=160&issue=55403&date=20110512&spage=12&pages=12-12&title=New%20York%20Times&atitle=How%20to%20Survive%20A%20Clickjack%20Attack.&bttitle=New%20York%20Times&jtitle=New%20York%20Times&series=&aulast=Biersdorfer%2C%20J.%20D.&id=DOI:>]. Accessed 2021-01-19.
- [20] Clarke, N. 2011. Transparent User Authentication: Biometrics, RFID and Behavioural Profiling. Springer-Verlag London Limited, published online. [<https://link-springer-com.ludwig.lub.lu.se/content/pdf/10.1007%2F978-0-85729-805-8.pdf>]. Transparent User Authentication. Accessed 2021-03-23.
- [21] Hell, M. 2020. EITA25 Computer Security: User Authentication. [https://www.eit.lth.se/fileadmin/eit/courses/eita25/lect/EITA25_Lect4_User_Authentication.pdf]. Accessed 2021-03-23.
- [22] Ometov, A. Bezzateev, S. Mäkitalo, N. Andreev, S. Mikkonen, T. Koucheryavy, Y. 2018. Multi-Factor Authentication: A Survey. [<https://www.mdpi.com/2410-387X/2/1/1/htm>]. Accessed 2021-03-23.
- [23] Dinei, F. Cormac, H. 2007. A large-scale study of web password habits. [<https://dl.acm.org/doi/10.1145/1242572.1242661>]. Accessed 2021-03-23.
- [24] Dipankar, D. Arunava, R. Abhijit, N. 2016. Toward the design of adaptive selection strategies for multi-factor authentication. [<https://www.sciencedirect.com/science/article/pii/S016740481630102X>]. Accessed 2021-03-23.
- [25] Bonneau, J. Herley, C. 2015. Passwords and the evolution of imperfect authentication. [<https://dl.acm.org/doi/abs/10.1145/2699390>]. Accessed 2021-03-23.
- [26] Krishnan Konoth, R. van der Veen, V. Bos, H. 2017. How Anywhere Computing Just Killed Your Phone-Based Two-Factor Authentication. [https://link.springer.com/chapter/10.1007/978-3-662-54970-4_24#Fn1]. Accessed 2021-03-24.
- [27] Nancie, G. Diarmid, M. Hazel, M. Mervyn J. 2011. User perceptions of security and usability of single-factor and two-factor authentication in automated telephone banking. [<https://www.sciencedirect.com/science/article/pii/S0167404810001148>]. Accessed 2021-03-28.
- [28] Sun, J. Zhang, R. Zhang, J. Zhang, Y. 2014. TouchIn: Sightless two-factor authentication on multi-touch mobile devices. [<http://resolver.ebscohost.com.ludwig.lub.lu.se/openurl?sid=EBSCO%3aedseee&genre=chapter&issn=edseee.IEEEConferenc&isbn=9781479958900&volume=&issue=&date=20141001&spage=436&pages=436-444&title=2014+IEEE+Conference+on+Communications+and>

- Network+Security%2c+Communications+and+Network+Security+(CNS)%2c+2014+IEEE+Conference+on&atitle=TouchIn%3a+Sightless+two-factor+authentication+on+multi-touch+mobile+devices&btile=2014+IEEE+Conference+on+Communications+and+Network+Security%2c+Communications+and+Network+Security+(CNS)%2c+2014+IEEE+Conference+on&jtitle=2014+IEEE+Conference+on+Communications+and+Network+Security%2c+Communications+and+Network+Security+(CNS)%2c+2014+IEEE+Conference+on&series=&aulast=Jingchao+Sun&id=DOI%3a10.1109%2fCNS.2014.6997513&site=ftf-live]. Accessed 2021-03-28.
- [29] Kishore Kumar, K. Deepthishree A. M. 2019. Comparison-Based Analysis of Different Authenticators. [https://link-springer-com.ludwig.lub.lu.se/content/pdf/10.1007%2F978-981-13-0212-1_59.pdf]. Accessed 2021-03-28.
- [30] Bruun, A. Jensen, K. Kristensen, D. 2014. Usability of Single- and Multi-factor Authentication Methods on Tabletops: A Comparative Study. [https://link-springer-com.ludwig.lub.lu.se/chapter/10.1007%2F978-3-662-44811-3_22]. Accessed 2021-03-28.
- [31] Grassi, P. Fenton, J. Newton, E. Perlner, R. Regenscheid, A. Burr, W. Richer, J. Lefkowitz, N. Danker, J. Choong, Y. Greene, K. Theofanos, M. 2017. Digital Identity Guidelines: Authentication and Lifecycle Management. NIST Special Publication 800-63B. [<https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-63b.pdf>]. Accessed 2021-03-28.
- [32] Klang, M. 2020. Database Technology: SQL. [<https://fileadmin.cs.lth.se/cs/Education/EDAF20/2020/lectures/f3/13.pdf>]. Accessed 2021-03-28.
- [33] Reese, K. Smith, T. Dutson, J. Armknecht, J. Cameron, J. Seamons, K. 2019. A Usability Study of Five Two-Factor Authentication Methods. [<https://www.usenix.org/system/files/soups2019-reese.pdf>]. Accessed 2021-03-28.
- [34] Bhagavatula, R. Ur, B. Iacovino, K. Kywe, S. Cranor, L. 2015. Biometric authentication on iPhone and Android: Usability, perceptions, and influences on adoption. [https://ink.library.smu.edu.sg/cgi/viewcontent.cgi?article=4969&context=sis_research]. Accessed 2021-03-28.
- [35] Wimberly, H. Liebrock, L. 2011. Using Fingerprint Authentication to Reduce System Security: An Empirical Study. [[http://resolver.ebscohost.com.ludwig.lub.lu.se/openurl?sid=EBSCO%3aedsee&genre=chapter&issn=10816011&isbn=9781457701474&volume=&issue=&date=20110501&spage=32&pages=32-46&title=2011+IEEE+Symposium+on+Security+and+Privacy%2c+Security+and+Privacy+\(SP\)%2c+2011+IEEE+Symposium+on&atitle=Using+Fingerprint+Authentication+to+Reduce+System+Security%3a+An+Empirical+Study&btile=2011+IEEE+Symposium+on+Security+and+Privacy%2c+Security+and+Privacy+\(SP\)%2c+2011+IEEE+Symposium+on&jtitle=2011+IEEE+Symposium+on+Security+and+](http://resolver.ebscohost.com.ludwig.lub.lu.se/openurl?sid=EBSCO%3aedsee&genre=chapter&issn=10816011&isbn=9781457701474&volume=&issue=&date=20110501&spage=32&pages=32-46&title=2011+IEEE+Symposium+on+Security+and+Privacy%2c+Security+and+Privacy+(SP)%2c+2011+IEEE+Symposium+on&atitle=Using+Fingerprint+Authentication+to+Reduce+System+Security%3a+An+Empirical+Study&btile=2011+IEEE+Symposium+on+Security+and+Privacy%2c+Security+and+Privacy+(SP)%2c+2011+IEEE+Symposium+on&jtitle=2011+IEEE+Symposium+on+Security+and+)

- Privacy%2c+Security+and+Privacy+(SP)%2c+2011+IEEE+Symposium+on&series=&aulast=Wimberly%2c+H.&id=DOI%3a10.1109%2fSP.2011.35&site=ftf-live]. Accessed 2021-03-28.
- [36] Wikipedia, The Free Encyclopedia. 2021. Fingerprint, Minutiae features. [https://en.wikipedia.org/wiki/Fingerprint#Minutiae_features]. Accessed 2021-03-29.
- [37] Malladi, S. Alves-Foss, J. Hechendorf, R. 2002. On Preventing Replay Attacks on Security Protocols. [<https://apps.dtic.mil/dtic/tr/fulltext/u2/a462295.pdf>]. Accessed 2021-03-30.
- [38] Harini, N. Padmanabhan, T.R. 2013. 2CAuth: A New Two Factor Authentication Scheme Using QR-Code. [<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.411.9555&rep=rep1&type=pdf>]. Accessed 2021-04-12.
- [39] Grasse, P. Garcia, M. Fenton, J. 2017. NIST Special Publication 800-63-3: Digital Identity Guidelines. [<https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-63-3.pdf>]. Accessed 2021-04-12.
- [40] NIST. Dworkin, M. 2015. NIST Policy on Hash Functions. [<https://csrc.nist.gov/Projects/Hash-Functions/NIST-Policy-on-Hash-Functions>]. Accessed 2021-04-13.
- [41] Wikipedia. 2021. SHA-1. [https://en.wikipedia.org/wiki/SHA-1#Comparison_of_SHA_functions]. Accessed 2021-04-13.
- [42] Aravindhana, K. Karthiga, R. 2013. One-time Password: A Survey. [https://www.researchgate.net/profile/Aravindhana-Kurunthachalam/publication/344518837_One-time_Password_A_Survey/links/5f7dd49592851c14bcb60412/One-time-Password-A-Survey.pdf]. Accessed 2021-04-13.
- [43] Jover, R. 2020. Security Analysis of SMS: as a Second Factor of Authentication. [<https://dl.acm.org/doi/pdf/10.1145/3424302.3425909>]. Accessed 2021-04-19.

The OAuth 2.0 Authorization Framework

This Appendix expands the description of OAuth 2.0 in Chapter 3 and gives more detailed information about the authorization code flow, the implicit flow, the resource owner password credentials flow, and the client credentials flow of the OAuth 2.0 Authorization Framework.

A.1 Authorization Code Flow

This section describes the purpose of the messages 1, 3, 4, and 5 in Figure 3.1 of the authorization code flow.

A.1.1 Authorization Request, Message 1

The "application/x-www-form-urlencoded" format is used for the authorization request, sent from the client to the authorization endpoint at the authorization server. The following parameters are included in the authorization request:

//REQUIRED PARAMETERS:

response_type

Value must be set to "code"

client_id

The client identifier, see Section 3.1 for more information

//OPTIONAL PARAMETERS:

redirect_uri

An absolute URI to which the authorization endpoint will send the user-agent back to once access is granted or denied

scope

A list of space-delimited, case-sensitive strings that specifies the scope of the request

//RECOMMENDED PARAMETERS:

state

An opaque value that is used by the client to maintain a state between the request and the callback. This value is included, by the authorization server,

when redirecting the user-agent back to the client. The parameter should be used for preventing cross-site request forgery. See Section 6.1 for more information

Example of an authorization request, a HTTP GET request, using TLS:

```
GET /authorize?response_type=code&client_id=s6BhdRkqt3
&state=xyz&redirect_uri=https%3A%2F%2Fclient
%2Eexample%2Ecom%2Fcb HTTP/1.1
Host: server.example.com
```

A.1.2 Authorization Response, Message 3

An authorization response is sent if the resource owner grants access in message 2. The "application/x-www-form-urlencoded" format is used for the authorization response, sent from the authorization endpoint at the authorization server to the client. The following parameters are included in the authorization response:

//REQUIRED PARAMETERS

code

The authorization code is generated by the authorization server and must expire shortly after it is issued to mitigate the risk of leaks, a maximum lifetime of 10 minutes is recommended.

The authorization code must not be used more than once and the authorization code is bound to the client identifier and the redirection URI.

state //REQUIRED IF PRESENT IN REQUEST

The value that was received in the request from the client

Example of an authorization response, a HTTP response:

```
HTTP/1.1 302 Found
Location: https://client.example.com/cb?
code=SplxlOBeZQQYbYS6WxSbIA&state=xyz
```

A.1.3 Error Response, Message 3

An error response is sent if the resource owner denies access in message 2 or if the request fails for reasons other than a missing or invalid redirection URI. Information should be sent to the resource owner from the authorization server if the request fails due to a missing or invalid client identifier, or a missing, invalid, or mismatching redirection URI. The authorization server must not redirect the user-agent to the invalid or mismatching redirection URI.

The "application/x-www-form-urlencoded" format is used for the error response, sent from the authorization endpoint at the authorization server to the client. The following parameters are included in the error response:

```
//REQUIRED PARAMETERS
error
    An ASCII error code form the following:
        invalid_request
        unauthorized_client
        access_denied
        unsupported_response_type
        invalid_scope
        server_error
        temporarily_unavailable
state //REQUIRED IF PRESENT IN REQUEST
    The value that was received in the request from the
    client
//OPTIONAL PARAMETERS
error_description
    Human-readable ASCII with additional information
error_uri
    URI to a web page with human-readable information about
    the error
```

Example of an error response, a HTTP response, where the authorization redirects the user-agent to the redirection URI:

```
HTTP/1.1 302 Found
Location: https://client.example.com/cb?error=access_denied
        &state=xyz
```

A.1.4 Access Token Request, Message 4

The "application/x-www-form-urlencoded" format is used for the token request, sent from the client to the token endpoint at the authorization server. The following parameters are included in the token request:

```
//REQUIRED PARAMETERS
grant_type
    Value must be set to "authorization_code"
code
    The authorization code that was received from the
    authorization server
redirect_uri
    The redirect URI that was included in the
    authorization request, the value
    must be identical to the value in the authorization
    request, see Section A.1.1 for more information
client_id //REQUIRED IF NO CLIENT AUTHENTICATION
    See Section 3.1 for more information
```

Example of an authorization request, a HTTP GET request, using TLS:

```
POST /token HTTP/1.1
Host: server.example.com
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded
```

```
grant_type=authorization_code
&code=SplxlOBeZQQYbYS6WxSbIA
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
```

The authorization server must require client authentication for confidential clients and authenticate the client if client authentication is included. The authorization server must also ensure that the authorization code was issued to the authenticated confidential client/the "client_id" for the public client, as well as verify that the authorization code is valid. If the "redirect_uri" was present as a parameter in the authorization request, the authorization server must verify that the value in the token request is identical to the value provided in the authorization request (see Section A.1.1).

A.1.5 Access Token Response, Message 5

An access token response containing an access token and, optionally, a refresh token is sent if the access token request is valid and authorized. Otherwise, an error response is sent. See Section 3.3.2 for more details.

An example of a successful access token response:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "access_token": "2YotnFZFEjr1zCsicMWpAA",
  "token_type": "example",
  "expires_in": 3600,
  "refresh_token": "tGzv3JOkF0XG5Qx2TlKWIA",
  "example_parameter": "example_value"
}
```

A.2 Implicit Flow

This section describes the purpose of the messages 1 and 3 in Figure 3.2 of the authorization code flow.

A.2.1 Authorization Request, Message 1

The "application/x-www-form-urlencoded" format is used for the authorization request, sent from the client to the authorization endpoint at the authorization

server. The following parameters are included in the authorization request:

```
//REQUIRED PARAMETERS:
response_type
    Value must be set to "token"
client_id
    The client identifier, see Section 3.1 for more
    information
//OPTIONAL PARAMETERS:
redirect_uri
    An absolute URI to which the authorization endpoint
    will send the user-agent back to once access is
    granted or denied
scope
    A list of space-delimited, case-sensitive strings
    that specifies the scope of the request
//RECOMMENDED PARAMETERS:
state
    An opaque value that is used by the client to
    maintain a state between the request and the
    callback. This value is included, by the
    authorization server, when redirecting the
    user-agent back to the client. The parameter
    should be used for preventing cross-site request
    forgery. See Section 6.1 for more information
```

Example of an authorization request, a HTTP GET request, using TLS:

```
GET /authorize?response_type=token&client_id=s6BhdRkqt3
&state=xyz&redirect_uri=https%3A%2F%2Fclient
%2Eexample%2Ecom%2Fcb HTTP/1.1
Host: server.example.com
```

A.2.2 Access Token Response, Message 3

An access token response containing an access token is redirected to the client if the resource owner grants the access request. Otherwise, an error response is sent. The "application/x-www-form-urlencoded" format is used for the access token response.

The following parameters are added to the fragment component of the redirection URI:

```
//REQUIRED PARAMETERS
access_token
    The access token that is issued by the authorization
    server
token_type
    The type of the token that is issued by the
```



```

    authorization server, examples of access token types
    are "bearer" and "mac"
state //REQUIRED IF PRESENT IN REQUEST
    The value that was received in the request from the
    client
//RECOMMENDED PARAMETERS
expires_in
    The access token's lifetime in seconds, from the
    time that the response was generated
//OPTIONAL PARAMETERS
scope //REQUIRED IF NOT IDENTICAL TO REQUEST
    A list of space-delimited, case-sensitive strings
    that specifies the scope of the request

```

An example of a successful access token response, that redirects the user-agent:

```

HTTP/1.1 302 Found
Location: http://example.com/cb
    #access_token=2YotnFZFEjr1zCsicMWpAA&state=xyz
    &token_type=example&expires_in=3600

```

A.2.3 Error Response, Message 3

An error response is sent if the resource owner denies access in message 2 or if the request fails for reasons other than a missing or invalid redirection URI. Information should be sent to the resource owner from the authorization server if the request fails due to a missing or invalid client identifier, or a missing, invalid, or mismatching redirection URI. The authorization server must not redirect the user-agent to the invalid or mismatching redirection URI.

The "application/x-www-form-urlencoded" format is used for the error response, sent from the authorization endpoint at the authorization server to the client. The following parameters are included in the error response:

```

//REQUIRED PARAMETERS
error
    An ASCII error code form the following:
        invalid_request
        unauthorized_client
        access_denied
        unsupported_response_type
        invalid_scope
        server_error
        temporarily_unavailable
state //REQUIRED IF PRESENT IN REQUEST
    The value that was received in the request from the
    client
//OPTIONAL PARAMETERS

```

```

error_description
    Human-readable ASCII with additional information
error_uri
    URI to a web page with human-readable information
    about the error

```

Example of an error response, a HTTP response, where the authorization redirects the user-agent to the redirection URI:

```

HTTP/1.1 302 Found
Location: https://client.example.com/cb
        #error=access_denied&state=xyz

```

A.3 Resource Owner Password Credentials Flow

This section describes the purpose of the messages in Figure 3.3 of the resource owner password credentials flow.

A.3.1 Authorization Request and Response, Message 1

The client must discard the resource owner credentials once the access token has been received. The method through which the client obtains the resource owner credentials is beyond the scope of this protocol.

A.3.2 Access Token Request, Message 2

The "application/x-www-form-urlencoded" format is used for the token request, sent from the client to the token endpoint at the authorization server. The following parameters are included in the token request:

```

//REQUIRED PARAMETERS
grant_type
    Value must be set to "password"
username
    The resource owner username
password
    The resource owner password
//OPTIONAL PARAMETERS
scope
    A list of space-delimited, case-sensitive strings
    that specifies the scope of the request

```

Example of an access token request, a HTTP POST, using TLS:

```

POST /token HTTP/1.1
Host: server.example.com
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded

```

```
grant_type=password&username=johndoe&password=A3ddj3w
```

The authorization server must require client authentication for confidential clients (or any client that was issued client credentials) and authenticate the client if client authentication is included. The authorization server must also validate the resource owner password credentials using its existing password validation algorithm.

The access token request includes the resource owner's password, which means that it is vulnerable to brute force attacks, see Section 6.2. Therefore, the authorization server must protect against brute force attacks.

A.3.3 Access Token Response, Message 3

An access token response containing an access token and, optionally, a refresh token is sent if the access token request is valid and authorized. Otherwise, an error response is sent. See Section 3.3.2 for more details.

An example of a successful access token response:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "access_token": "2YotnFZFEjr1zCsicMWpAA",
  "token_type": "example",
  "expires_in": 3600,
  "refresh_token": "tGzv3JOkF0XG5Qx2TlKWIA",
  "example_parameter": "example_value"
}
```

A.4 Client Credentials Flow

This section describes the purpose of the messages in Figure 3.4 of the client credentials flow.

A.4.1 Access Token Request, Message 1

The "application/x-www-form-urlencoded" format is used for the token request, sent from the client to the token endpoint at the authorization server. The following parameters are included in the token request:

```
//REQUIRED PARAMETERS
grant_type
    Value must be set to "client_credentials"
//OPTIONAL PARAMETERS
scope
```

A list of space-delimited, case-sensitive strings that specifies the scope of the request

Example of an access token request, a HTTP POST, using TLS:

```
POST /token HTTP/1.1
Host: server.example.com
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded
```

```
grant_type=client_credentials
```

The authorization server must authenticate the client.

A.4.2 Access Token Response, Message 2

An access token response containing an access token is sent if the access token request is valid and authorized. Otherwise, an error response is sent. See Section 3.3.2 for more details. A refresh token should not be included.

An example of a successful access token response:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Cache-Control: no-store
Pragma: no-cache
```

```
{
  "access_token": "2YotnFZFEjr1zCsicMWpAA",
  "token_type": "example",
  "expires_in": 3600,
  "example_parameter": "example_value"
}
```

OpenID Connect 1.0

This Appendix expands the description of OpenID in Chapter 4 and gives more detailed information about the authorization code flow, the implicit flow, and the hybrid flow of OpenID Connect 1.0.

B.1 Authorization Code Flow

This section describes the purpose of the messages in Figure 4.1, the endpoints, and the ID and access tokens of the authorization code flow.

B.1.1 Endpoints

There are three endpoints at the authorization server in the authorization code flow, the *authorization* endpoint, the *token* endpoint and the *userinfo* endpoint (see Section 4.6).

Authorization Endpoint

The authorization endpoint at the authorization server performs authentication and authorization of the end-user, by sending the user-agent to the authorization endpoint. Communication with the authorization endpoint must utilize TLS.

Token Endpoint

The token endpoint at the authorization server receives token requests and issues token responses to clients (RPs). Communication with the token endpoint must utilize TLS.

B.1.2 ID Token

The additional claim for the ID token when using the authorization code flow is:

```
//OPTIONAL CLAIM:  
at_hash
```

A case sensitive string representing the access token hash value, where the hash algorithm used is the hash algorithm used in the "alg" header parameter of the ID token's JOSE header, the value is the base64url encoding of the left-most half of the hash of the octets of the ASCII representation of the "access_token" value

ID Token Validation

The clients (RPs) must validate the ID token in the token response from the token endpoint according to the following:

1. If the ID token is encrypted, decryption of the ID token is done using the keys and algorithm specified during registration.
2. The "iss" value must be identical to the issuer identifier of the authorization server (OP), usually obtained during discovery.
3. The "aud" claim must contain the "client_id" value present in the "iss" claim. The "aud" claim may contain more than one value. However, the ID token must be rejected if the audience claim contains audiences that are not trusted by the client.
4. The "azp" claim should be present if the ID token contains multiple audiences.
5. If the "azp" claim is present, the "client_id" should be its claim value.
6. The LST server may be used for validating the issuer instead of checking the token signature. However, the signature of all other ID tokens must be validated by the client according to JWS (using the algorithm specified in the JWT "alg" header parameter). The client must use the keys provided by the issuer.
7. The "alg" value should be the default of "RS256" or the algorithm value in the "id_token_signed_response_alg" parameter that is sent during registration.
8. If the "alg" parameter is using a MAC based algorithm (e.g, "HS256", "HS384" or "HS512"), the octets of the UTF-8 representation of the "client_secret" corresponding to the "client_id" contained in the "aud" claim are used as the key to validate the signature. The behaviour is unspecified if the "aud" is multi-valued or if the "azp" value is present that is different than the "aud" value.
9. The current time must be before the time that is specified in the "exp" claim.
10. The time-value in the "iat" claim can be used to reject ID tokens that were issued too far away from the current time. By limiting the time that the ID tokens are valid, the time that the "nonce" value is stored is also limited, which prevents attacks.

11. The value of the "nonce" claim must be present and identical to the authentication request's "nonce" parameter, if the "nonce" parameter was present in the request. The "nonce" value should be checked for replay attacks (see Section 6.3).
12. The asserted claim value of the "acr" claim should be checked if present.
13. The "auth_time" claim should be checked if requested (through a specific request or through the "max_age" parameter), to determine if too much time has past since the last end-user authentication.

B.1.3 Access Token Validation

If the ID token contains an "at_hash" claim, the access token and the id token may be validated by the client (RP) as defined in Section B.2.3 (but in this code flow the access token and the id token are returned from the token endpoint).

B.1.4 Authentication Request, Message 2

An authentication request is an OAuth 2.0 authorization request, which requests authentication of the end-user at the authorization endpoint at the authorization server. The following parameters are included in the request:

//REQUIRED PARAMETERS:

scope

The "openid" value must be included in the scope parameter, other scopes may be included, see Section 4.3 for more details

response_type

The value is "code"

client_id

The OAuth 2.0 client identifier

redirect_uri

The redirection uri to which the response will be sent, must exactly match one of the redirection uri values that are pre-registered at the openid provider, redirection uri should use https scheme (may use http if client is confidential)

//RECOMMENDED PARAMETERS:

state

An opaque value that represents the state between the request and the callback

//OPTIONAL PARAMETERS:

response_mode

Informational parameter to the authorization server about the mechanism to be used for returning parameters from the authorization endpoint, not recommended if response mode is the default mode specified for the response_type

nonce
String that associates a client session with an ID token and mitigates replay attacks, unmodified from the authentication request to the id token

display
ASCII string value, specifies how the authorization server displays the authentication and consent user interface pages to the end user, the defined values are:
page
popup
touch
wap

prompt
Space delimited, case sensitive list of ASCII string values, specifies whether the authorization server prompts the end-user for reauthentication and consent, the defined values are:
none
authorization server must not display any authentication or consent user interface page, will return an error if, for example, the end-user is not authenticated, can be used to check for existing authentication and/or consent
login
authorization server should prompt end-user for reauthentication
consent
authorization server should prompt end-user for consent before returning information to the client
select_account
authorization server should prompt the end-user to select a user account

max_age
maximum authentication age in seconds since the last time the end-user was actively authenticated by the authorization server, if time is elapsed the end-user must reauthenticate

ui_locales
end-user's preferred languages and scripts for the user interface (language tag values)

id_token_hint
hint about the id token previously issued by the authorization server, about the end-user's current or past authenticated session with the client

login_hint

hint about the login identifier, e.g., email or phone number, the end-user might use to log in

acr_values

authentication context class reference values that are requested, specifies the acr values that the authorization server is being requested to use for processing this authentication request

Example of an HTTP 302 redirect response by the client, followed by a HTTP GET request sent by the user-agent to the authorization server in response to the HTTP 302 redirect response by the client:

HTTP/1.1 302 Found

Location: [https://server.example.com/authorize?](https://server.example.com/authorize?response_type=code&scope=openid%20profile%20email&client_id=s6BhdRkqt3&state=af0ifjsldkj&redirect_uri=https%3A%2F%2Fclient.example.org%2Fc)

response_type=code

&scope=openid%20profile%20email

&client_id=s6BhdRkqt3

&state=af0ifjsldkj

&redirect_uri=https%3A%2F%2Fclient.example.org%2Fc

GET /authorize?

response_type=code

&scope=openid%20profile%20email

&client_id=s6BhdRkqt3

&state=af0ifjsldkj

&redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb HTTP/1.1

Host: server.example.com

Authentication Request Validation

The authorization server must: validate the OAuth 2.0 parameters according to the OAuth 2.0 specification, verify that a scope parameter is present with the openid value, verify that all the required parameters are present and used correctly, and only send a positive response when sub claim is requested with a specific value for the ID token (in, for example, the "id_token_hint" parameter) if the end-user identified by that sub value has an active session with the authorization server or has been authenticated.

B.1.5 Authorization Server Authenticates End-User

If the authentication request is valid, the authorization server authenticates the end-user (or determines if the end-user already is authenticated). The methods used for authentication by the authorization server is beyond the scope of this specification. If the authentication request contains the "prompt" parameter with the value "login", the authorization server must authenticate the end-user even if the end-user already is authenticated. If the "prompt" parameter has the value

"none", the authorization server must return an error if an end-user is already authenticated. The authorization server must employ appropriate measures against cross-site request forgery and clickjacking, see Sections 6.1 and 6.4, respectively.

B.1.6 Authorization Server Obtains End-User Consent/Authorization

Before releasing information to the client (RP), the authorization server must obtain an authorization decision from the end-user after the end-user has been authenticated.

B.1.7 Authentication Response, Message 3

An authentication response is an OAuth 2.0 authorization response that is sent from the authorization endpoint at the authorization server (OP) to the client (RP). If the authentication request sent is valid, end-user has been authenticated and has given consent and authorization, a successful authentication response is sent (see Section B.1.7). Else, an authentication error response is sent (see Section B.1.7).

Successful Authentication Response

The "application/x-www-form-urlencoded" format is used for the authorization response, sent from the authorization endpoint at the authorization server (OP) to the client (RP). The parameters in the authentication response are defined in the OAuth 2.0 protocol, see Section A.1.2, and added as query parameters to the "redirect_uri" (unless specified otherwise). Example of an authorization response, a HTTP response:

```
HTTP/1.1 302 Found
Location: https://client.example.org/cb?
        code=SplxlOBeZQQYbYS6WxSbIA
        &state=af0ifjlsldkj
```

Authentication Error Response

If the end-user denies the authentication request or if the authentication of the end-user fails, an authentication error response is sent from the authorization server (OP) to the client (RP). The parameters in the authentication response are defined in the OAuth 2.0 protocol, see Section A.1.3. If the redirection URI is valid, the authorization server returns the client to the redirection URI specified in the authorization request. The following parameters are additional error codes to the OAuth 2.0 protocol and are specified by the OpenID Connect specification:

```
interaction_required
login_required
account_selection_required
consent_required
invalid_request_uri
```

```
invalid_request_object
request_not_supported
request_uri_not_supported
registration_not_supported
```

Example of an authentication error response:

```
HTTP/1.1 302 Found
Location: https://client.example.org/cb?
    error=invalid_request
    &error_description=Unsupported%20response_type%20value
    &state=af0ifjsldkj
```

B.1.8 Token Request, Message 4

The parameters of a token request are defined by the OAuth 2.0 protocol, see Section A.1.4. If the client is of client type confidential client, the client must authenticate to the token endpoint using the authentication method registered for its "client_id", see Section 4.4.

Example of a Token Request, using HTTP POST:

```
POST /token HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW

grant_type=authorization_code&code=
    SplxlOBeZQQYbYS6WxSbIA
&redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
```

Token Request Validation

The authorization server must: authenticate the client if it was issued client credentials or if it uses another client authentication method according to Section 4.4, ensure that the authorization code was issued to the authenticated client, verify and validate the authorization code and check that it has not been previously used, verify that the "redirect_uri" parameter is identical to the one in the authentication request, and verify that the authorization code was issued in response to an OpenID Connect Authentication Request.

B.1.9 Token Response, Message 5

A token response is sent from the token endpoint at the authorization server (OP) to the client (RP). If the token request sent is valid and authorized, a successful token response is sent (see Section B.1.9). Else, an token error response is sent (see Section B.1.9).

Successful Token Response

The "application/json" format is used for the token response, sent from the token endpoint at the authorization server (OP) to the client (RP). The parameters in the token response are defined in the OAuth 2.0 protocol, see Section A.1.5, and additional parameters specified by the OpenID Connect protocol are:

```
//REQUIRED PARAMETERS:
id_token
    ID token value associated with the authenticated
    session
```

Example of a token response, a HTTP response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store
Pragma: no-cache

{
  "access_token": "SIAV32hkKG",
  "token_type": "Bearer",
  "refresh_token": "8xLOxBtZp8",
  "expires_in": 3600,
  "id_token": "eyJhbGciOiJSUzI1NiIsImtpZCI6IjF1ZWdka3cifQ.ewogImlzcyI6ICJodHRwOi8u...EEBLuVVk4XUVrWOLrLl0nx7RkKU8NXNHq-rvKMzqg"
}
```

Token Error Response

If the token request is invalid or unauthorized a token error response is sent from the authorization server (OP) to the client (RP). The parameters in the authentication response are defined in the OAuth 2.0 protocol, see Section 3.3.2. The "application/json" format is used for the token response.

Example of an authentication error response:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Cache-Control: no-store
Pragma: no-cache

{
  "error": "invalid_request"
}
```

Token Response Validation

The client must: follow the validation rules specified in the OAuth 2.0 protocol, follow the ID token and access token validation rules, see Section B.1.2 and B.1.3, respectively.

B.2 Implicit Flow

This section describes the purpose of the messages in Figure 4.2, the endpoints, and the ID and access tokens of the implicit flow.

B.2.1 Endpoints

There are two endpoints at the authorization server in the implicit flow, the authorization endpoint and the userinfo endpoint (see Section 4.6).

Authorization Endpoint

Used in the same manner as for the authorization code flow, see Section B.1.1, except for some details explained below.

B.2.2 ID Token

The additional requirements for claims of the ID token in the implicit flow defined in this protocol are:

```
//REQUIRED PARAMETERS
```

```
nonce
```

```
at_hash //REQUIRED IF "response_type" VALUE IS "id_token  
token" IN REQUEST
```

```
  A case sensitive string representing the access token  
  hash value, where the hash algorithm used is the  
  hash algorithm used in the "alg" header parameter  
  of the ID token's JOSE header, the value is the  
  base64url encoding of the left-most half of the  
  hash of the octets of the ASCII representation of  
  the "access_token" value
```

ID Token Validation

The ID token validation must be done in the same manner in the implicit flow as for the authorization code flow, see Section B.1.2, except for some details given in this section.

The client must validate the signature of the ID token according to JWS (by using the algorithm specified in the "alg" header parameter of the JOSE header). The value of the "nonce" claim must be identical to the value in the authentication request if present. The "nonce" value should be checked for replay attacks.

B.2.3 Access Token Validation

The client should hash the octets of the ASCII representation of the "access_token" according to JWA (by using the algorithm specified in the "alg" header parameter of the JOSE header), the left-most half of the hash should be base64url encoded and the encoded value must match the "at_hash" value of the ID token.

B.2.4 Authentication Request, Message 2

The authentication request is sent from the client to the authorization endpoint at the authorization server. The parameters in the authentication request are defined in Section B.1.4, except for the following parameters:

//REQUIRED PARAMETERS:

response_type

The value is "id_token token" (ID token and access token will be returned) or "id_token" (ID token will be returned)

redirect_uri

The redirection URI to which the response will be sent, must match on of the redirection URI values for the client pre-registered at the OpenID provider, must not use the "http" scheme unless the client is a native application that uses "localhost" as the hostname

nonce

String value that associates a client session with an ID token and mitigates replay attacks, unmodified from the authentication request to the id token

An example of an authentication request, a HTTP GET request, sent by the user-agent to the authorization server in response to a HTTP 302 redirect response by the client looks as follows:

```
GET /authorize?
  response_type=id_token%20token
  &client_id=s6BhdRkqt3
  &redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
  &scope=openid%20profile
  &state=af0ifjlsldkj
  &nonce=n-0S6_WzA2Mj HTTP/1.1
Host: server.example.com
```

Authentication Request Validation

The authentication request is validated in the same manner in the implicit flow as for the authorization code flow, see Section B.1.4.

B.2.5 Authorization Server Authenticates End-User

The authorization server authenticates the end-user in the same manner in the implicit flow as for the authorization code flow, see Section B.1.5.

B.2.6 Authorization Server Obtains End-User Consent/Authorization

The authorization server obtains the end-user consent in the same manner in the implicit flow as for the authorization code flow, see Section B.1.6.

B.2.7 Authentication Response, Message 3

An authentication response is made in the same manner in the implicit flow as for the authorization code flow, see Section B.1.7, except for some details given in this section. If the authentication request sent is valid, end-user has been authenticated and has given consent and authorization, a successful authentication response is sent (see Section B.2.7). Else, an authentication error response is sent (see Section B.2.7).

Successful Authentication Response

A successful authentication response is sent from the authorization endpoint at the authorization server to the client. The response parameters in the authentication response are sent in the fragment component of the redirection URI, unless a different response mode was specified. The following parameters are included in the authentication response:

`access_token`

An access token is returned if the "response_type" had the value "id_token token" in the authorization request

`token_type`

This is returned if an access token is included in the authentication response, the value must be "Bearer", unless client and authorization server has specified otherwise

//REQUIRED PARAMETERS:

`id_token`

The ID token

`state` //REQUIRED IF PRESENT IN REQUEST

The OAuth 2.0 state value, client must verify that the value is identical to the value in the authorization request

//OPTIONAL PARAMETERS:

`expires_in`

The access token's lifetime in seconds, from the time that the response was generated

An example of an authorization response, a HTTP response may look like:


```
HTTP/1.1 302 Found
Location: https://client.example.org/cb#
    access_token=SIaV32hkKG
    &token_type=bearer
    &id_token=eyJ0...NiJ9.eyJ1c...I6IjIifX0.DeWt4Qu...
        ZXso
    &expires_in=3600
    &state=af0ifj5ldkj
```

Authentication Error Response

An authentication error response is made in the same manner in the implicit flow as for the authorization code flow, see Section B.1.7, except for some details given in this section.

The authorization server must return the error authorization response in the fragment component of the redirection URI (unless response mode is specified otherwise), if the end-user denies the request or the end-user authentication fails.

Redirect URI Fragment Handling

The user-agent needs to be able to parse the fragment encoded values and pass them on to the client's processing logic for consumption, since the response parameters of the authentication response are returned in the redirection URI fragment value.

Authentication Response Validation

The client must verify that the response is according to Section 5 of the OAuth 2.0 Multiple Response Type Encoding Practices [15], as well as follow the validation rules in the following Sections:

1. Access token response in the OAuth 2.0 protocol for the implicit flow, Section A.2.2
2. Cross-site request forgery, Section 6.1
3. ID token validation according to the OpenID Connect protocol for the implicit flow, Section B.2.2
4. If "response_type" is "id_token token", access token validation according to the OpenID Connect protocol for the implicit flow, Section B.2.3

B.3 Hybrid Flow

This section describes the purpose of the messages in Figure 4.3, the endpoints, and the ID and access tokens of the hybrid flow.

B.3.1 Endpoints

The endpoints are used in the same manner for the hybrid flow as for the authorization code flow, see Section B.1.1, except for some details explained in this section.

Authorization Endpoint

The authorization endpoint at the authorization server is used in the same manner for the hybrid flow as for the authorization code flow, see Section B.1.1, except for some details explained in Section B.3.5 and Section B.3.8.

Token Endpoint

The token endpoint at the authorization server is used in the same manner for the hybrid flow as for the authorization code flow, see Section B.1.1, except for some details explained in Sections B.3.9, B.3.10, B.3.2 and B.3.3.

B.3.2 ID Token

The contents of an ID token is defined differently for the hybrid flow, depending on if the ID token is returned from the authorization endpoint or both the authorization endpoint and the token endpoint.

ID Tokens returned from the Authorization Endpoint:

The additional requirements for claims of the ID token in the hybrid flow, returned from the authorization endpoint at the authorization server, are:

```
//REQUIRED PARAMETERS
nonce
//OPTIONAL PARAMETERS
at_hash //REQUIRED IF "response_type" VALUE IS "id_token
token" IN REQUEST
    A case sensitive string representing the access token
    hash value, where the hash algorithm used is the
    hash algorithm used in the "alg" header parameter
    of the ID token's JOSE header, the value is the
    base64url encoding of the left-most half of the
    hash of the octets of the ASCII representation of
    the "access_token" value
c_hash //REQUIRED IF "response_type" VALUE IS "code
id_token" OR "code id_token token"
    A case sensitive string representing the code hash
    value, where the hash algorithm used is the hash
    algorithm used in the "alg" header parameter of the
    ID token's JOSE header, the value is the base64url
    encoding of the left-most half of the hash of the
```

octets of the ASCII representation of the "
access_token" value

ID Tokens returned from the Authorization Endpoint and from the Token Endpoint:

- The ID token is returned from both the authorization endpoint and the token endpoint when "response_type" has the values "code id_token" and "code id_token token".
- The values of the claims "iss" and "sub" must be identical in both ID tokens.
- All claims about the authentication event should be present in both ID tokens.
- Any claims that are present in both ID tokens should have the same value in both ID tokens.
- The authorization server (OP) may choose to return fewer claims about the end-user from the authorization endpoint.
- The "at_hash" and "c_hash" claims may be omitted from the ID token returned from the token endpoint.

ID Token Validation

The ID tokens are validated differently depending on if the ID tokens are returned from the authorization endpoint or from the token endpoint. ID tokens that are returned from the authorization endpoint at the authorization server must be validated in the same manner in the hybrid flow as for the implicit flow, see Section B.2.2. ID tokens that are returned from the token endpoint must be validated in the same manner in the hybrid flow as for the authorization code flow, see Section B.1.2.

B.3.3 Access Token

When the value of "response_type" is "code token" or "code id_token token", the access token is returned from both the authorization endpoint and the token endpoint at the authorization server. Because of, for example, different security characteristics of the two endpoints, the values of the two access tokens may be the same or may be different, e.g., different lifetimes and different access to resources.

Access Token Validation

The access tokens are validated differently depending on if the access tokens are returned from the authorization endpoint or from the token endpoint. Access tokens that are returned from the authorization endpoint at the authorization server must be validated in the same manner in the hybrid flow as for the implicit flow, see Section B.2.3. Access tokens that are returned from the token endpoint must be validated in the same manner in the hybrid flow as for the authorization code flow, see Section B.1.3.

B.3.4 Authorization Code Validation

The client should hash the octets of the ASCII representation of the "code" according to JWA (by using the algorithm specified in the "alg" header parameter of the ID token's JOSE header), the left-most half of the hash should be base64url encoded and the encoded value must match the "c_hash" value of the ID token.

B.3.5 Authentication Request, Message 2

An authentication request is made in the same manner in the hybrid flow as for the authorization code flow, see Section B.1.4, except for the following parameters:

//REQUIRED PARAMETERS:

response_type

The value of this parameter determines which authorization processing flow that will be used, as well as what parameters will be returned from which endpoints, the values are "code id_token", "code token" or "code id_token token"

Example of an authentication request, a HTTP GET request, sent by the user-agent to the authorization server in response to a HTTP 302 redirect response by the client:

```
GET /authorize?
  response_type=code%20id_token
  &client_id=s6BhdRkqt3
  &redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
  &scope=openid%20profile%20email
  &nonce=n-0S6_WzA2Mj
  &state=af0ifjsldkj HTTP/1.1
Host: server.example.com
```

Authentication Request Validation

The authentication request is validated in the same manner in the hybrid flow as for the authorization code flow, see Section B.1.4.

B.3.6 Authorization Server Authenticates End-User

The authorization server authenticates the end-user in the same manner in the hybrid flow as for the authorization code flow, see Section B.1.5.

B.3.7 Authorization Server Obtains End-User Consent/Authorization

The authorization server obtains the end-user consent in the same manner in the hybrid flow as for the authorization code flow, see Section B.1.6.

B.3.8 Authentication Response, Message 3

An authentication response is made in the same manner in the hybrid flow as for the implicit flow, see Section B.2.7, except for some details given in this section. If the authentication request sent is valid, end-user has been authenticated and has given consent and authorization, a successful authentication response is sent (see Section B.3.8). Else, an authentication error response is sent (see Section B.3.8).

Successful Authentication Response

A successful authentication response is made in the same manner in the hybrid flow as for the implicit flow, see Section B.2.7, except for some details given in this section.

The following parameters are included in the authentication response from the authorization endpoint at the authorization server:

`access_token`
 The access token that is returned together with a "`token_type`" value when the value of "`response_type`" in the authentication request is "`code token`" or "`code id_token token`"

`id_token`
 The ID token that is returned when the value of "`response_type`" in the authentication request is "`code id_token`" or "`code id_token token`"

`code`
 The authorization code, always returned

Example of an authorization response, a HTTP response:

```
HTTP/1.1 302 Found
Location: https://client.example.org/cb#
  code=SplxlOBeZQQYbYS6WxSbIA
  &id_token=eyJ0...NiJ9.eyJ1c...I6IjIifX0.DeWt4Qu...
  ZXso
  &state=af0ifjlsldkj
```

Authentication Error Response

An authentication error response is made in the same manner in the hybrid flow as for the authorization code flow, see Section B.1.7, except for some details given in this section.

If the end-user denies the authentication request or if the authentication of the end-user fails, the authentication error response must be sent from the authorization server (OP) in the fragment component of the redirection URI.

Redirect URI Fragment Handling

The requirements for redirect URI fragment handling are the same in the hybrid flow as for the implicit flow, see Section B.2.7.

Authentication Response Validation

The client must verify that the response is according to Section 5 of the OAuth 2.0 Multiple Response Type Encoding Practices [15], as well as follow the validation rules in the following Sections:

1. Access token response in the OAuth 2.0 protocol for the implicit flow, Section A.2.2
2. Cross-site request forgery, Section 6.1
3. If "response_type" is "code id_token" or "code id_token token", ID token validation according to Section B.3.2
4. If "response_type" is "code token" or "code id_token token", access token validation according to Section B.3.3
5. If "response_type" is "code token" or "code id_token token", authorization code validation according to Section B.3.4

B.3.9 Token Request, Message 4

Token requests are made in the same manner in the hybrid flow as for the authorization code flow, see Section B.1.8.

Token Request Validation

Token requests are validated in the same manner in the hybrid flow as for the authorization code flow, see Section B.1.8.

B.3.10 Token Response, Message 5

A token response is sent from the token endpoint at the authorization server (OP) to the client (RP). If the token request sent is valid and authorized, a successful token response is sent (see Section B.3.10). Else, a token error response is sent (see Section B.3.10).

Successful Token Response

A token response is made in the same manner in the hybrid flow as for the authorization code flow, see Section B.1.9.

Token Error Response

A token error response is made in the same manner in the hybrid flow as for the authorization code flow, see Section B.1.9.

Token Response Validation

Token responses are validated in the same manner in the hybrid flow as for the authorization code flow, see Section B.1.9.



LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2021-828
<http://www.eit.lth.se>