Secure Token Management in Hybrid Applications

University of Turku
Department of Computing
Master of Science (Tech) Thesis
Software Engineering
April 2026
Eetu Pienimaa

Supervisors:
Sampsa Rauti (University of Turku)
Ville Leppänen (University of Turku)

UNIVERSITY OF TURKU
Department of Computing

Eetu Pienimaa: Secure Token Management in Hybrid Applications

Master of Science (Tech) Thesis, 65 p., 1 app. p.
Software Engineering
April 2026

This thesis examines the security of token management in hybrid applications that operate on web and mobile platforms. These applications commonly rely on OAuth 2.0 for authorization and OpenID Connect for authentication, where tokens function as credentials to access protected resources. Due to security differences between platforms, the secure management of tokens presents significant challenges.
This thesis addresses three research questions. It identifies the most critical threats to tokens, evaluates token storage mechanisms in terms of confidentiality and implementation cost, and proposes a secure and maintainable architecture. The results show that token exposure is the primary risk and high risk threats include cross-site scripting (XSS) and insecure storage. The storage mechanism evaluation shows a trade-off between confidentiality and implementation cost in hybrid applications. More confidential solutions, such as `HttpOnly` cookies require additional infrastructure and native secure storage APIs require additional effort to achieve similar security guarantees in the web environment. In contrast, simpler token storage mechanisms provide weaker confidentiality and protection. Finally, this thesis proposes an architecture that uses backend managed sessions for web environments and operating system backed secure storage for mobile platforms. This approach improves security, reduces token exposure and is practical to implement on both web and mobile platforms.

Keywords: Hybrid applications, OAuth 2.0, OIDC, authorization, authentication, Token, Token storage, Token management

Contents

List of Figures

List of Tables

1 Introduction

Modern applications are increasingly deployed on both web and mobile platforms. Hybrid application frameworks allow developers to build cross-platform applications by using a single codebase. This development practice can improve productivity and reduce costs, but at the same time it can introduce security challenges in management of secrets.

Many modern applications use OAuth 2.0 for authorization and OpenID Connect for authentication. In these applications, different types of tokens such as access tokens are used as credentials to gain access to protected resources. If these tokens are exposed or accidentally leaked, attackers may impersonate legitimate users and gain unauthorized access. Therefore, secure token management is a critical aspect of application security.

Browser-based and native mobile environments are inherently different ecosystems. They have different security characteristics and threat models. These differences make designing of a token management solution for hybrid applications more challenging. To address these challenges, this thesis examines token management in hybrid applications through three research questions. The research questions (RQ) are as follows:

1. **RQ1:** Which threats pose the greatest risk to tokens in hybrid applications?
2. **RQ2:** How do different token storage mechanisms compare in terms of confidentiality and implementation cost?

3. **RQ3:** How can a secure and maintainable token management solution be implemented for hybrid applications?

Before answering these questions, Chapter 2 introduces the background concepts, which include hybrid applications, authentication and authorization, OAuth 2.0, OpenID Connect and the security context of token management. Chapter 3 introduces the storage mechanisms available on web and mobile platforms and conducts threat modeling to answer **RQ1**. Chapter 4 evaluates and compares the storage mechanisms to answer **RQ2**. Chapter 5 answers **RQ3**, by proposing and implementing an architecture for hybrid applications. Finally, Chapter 6 concludes the thesis.

2 Background and Related Concepts

This chapter provides the conceptual and technological background for this thesis. It describes the underlying concepts, technologies and security challenges with token management in hybrid applications.

2.1 Hybrid Applications

Hybrid applications are software systems that allow a single codebase to be deployed across multiple platforms, such as in web browsers and mobile devices. Frameworks such as Apache Cordova [1], Expo [2], Ionic [3] and Flutter [4] have popularized this approach by offering developers the ability to "build once and run anywhere". Compared to maintaining separate native mobile applications on each platform and a separate web application, hybrid development provides benefits in the form of reduced development costs [5]. In addition to production-grade applications, hybrid frameworks are a suitable tool for fast prototyping.

Cross-platform development is a process of developing applications that run on multiple platforms [5]. In the past, developers had to build and maintain separate repositories for each platform. This required extensive knowledge of web technologies and platform-specific languages such as Kotlin on Android and Swift for iOS. Nowadays, with hybrid frameworks, the same result can be achieved with lower resources, since the core of hybrid applications is typically written by using well-known web technologies such as HTML, CSS and JavaScript [1], [2], [3]. This greatly sim-

plifies the development process, since there is no need to rely on multiple different technologies to deliver functionality comparable to applications made with native technologies.

In practice, hybrid applications do not differ significantly from native applications. Hybrid applications can access the same native APIs to use device-specific capabilities such as GPS, camera, speakers and storage utilities. In the past, there were concerns about their performance on mobile devices when being compared to native applications, but this issue has been actively worked on by the framework developers over the years, as seen by the latest revision of React Native's architecture [6], [7].

Depending on the framework, hybrid applications function differently at the system level on mobile devices. Some frameworks, such as Ionic, use iOS and Android `WebView` components, which allows the application to display web content on mobile devices and at the same time provide access to hardware functionalities. However, in some cases Ionic's architecture uses a bridge layer to access platform-specific APIs. Applications that are built with Ionic, the `WebView` is integrated through Capacitor, which is a native runtime container that allows web-based apps to run on different platforms.[8]

Flutter differs from the other previously mentioned frameworks in that it uses Dart as its primary language. Flutter's architecture communicates with the mobile device's native layer through a system called platform channels, which allows Dart code to send asynchronous messages to platform-specific code. When the application needs to use a native feature, it sends a message through the channel to the framework's embedder layer which forwards it to the native side for execution. The native platform then performs the operation and sends the result back through the same channel. [4], [9]

In React Native's latest version, its Native Module system introduces a JavaScript

Interface (JSI), which allows for both asynchronous and synchronous communication between the JavaScript and native layers. In its older architecture, React Native relied on an asynchronous bridge, which caused problems with native APIs that expect synchronous responses. [7]

This thesis focuses on Expo as the primary hybrid framework. Expo is an open-source platform built around React Native [2]. It was chosen due to its widespread adoption and active community. Additionally, React Native's latest architectural revision makes it a compelling option for developing hybrid applications with good performance.

Lastly, while hybrid applications offer several advantages for cross-platform development, they also introduce their own unique considerations. The abstraction of platform differences at the development level does not eliminate the fundamental differences between browsers and mobile devices. These differences become particularly visible in application's security related functionality, such as managing authentication processes and secrets.

2.2 Authorization and Authentication

Secure communication between applications and services relies on two closely related but different concepts: authorization and authentication. These terms are often used interchangeably, but they have different meanings in the context of system security. Before examining these concepts, it is important to first understand the broader principle that includes them both: access control.

Access control is defined as the enforcement of access policies [10]. It is a key component of information security that determines who has the right to access specific data, applications and resources. Access control works by verifying user's or a systems identity through credentials such as passwords, PINs, security tokens or biometric identifiers. Once the user's or a system's identity has been confirmed,

access control mechanisms apply predefined policies to grant the appropriate permissions. [11] There are four primary access control models that manage access in a distinct manner:

1. Discretionary access control (DAC): A resource owner grants access rights by specifying which users can access certain resources.

2. Role-based access control (RBAC): Permissions are related to specific roles within the system. Users are assigned to one or more roles, each of which grants access to specific resources.

3. Mandatory access control (MAC): Access rights are managed by a central authority, which assigns security levels to users and resources.

4. Attribute-based access control (ABAC): Access rights are granted dynamically based on user attributes and environmental factors, such as department, age or location. [11]

Authorization and authentication are relevant components of access control and they serve different purposes. Authorization can be defined as the specification of access policies, which means that users or systems are granted appropriate levels of access based on contextual factors such as their role. It determines what actions a given entity is able to perform within a system. In contrast, authentication is the process of verifying the user's or system's identity. It is used to ensure that the entity is truly who or what it claims to be. [10], [11], [12]

In modern systems, authorization is typically implemented by using the Open Authorization (OAuth) 2.0 [13] and authentication is handled by using the OpenID Connect (OIDC) identity layer [14]. These two protocols are not mutually exclusive. They are designed to work together to provide secure and reliable authorization and authentication mechanisms. For example, many hybrid frameworks, such as Expo

and Ionic, offer built-in packages that allow authorization and authentication to be implemented with various OAuth and OIDC providers [15], [16].

2.3 OAuth 2.0

OAuth 2.0 was published in 2012 in RFC 6749 to address the challenges of traditional client-server setups, where third-party applications need to directly handle the resource owner's credentials to access protected resources. These traditional models had several issues: applications had to store user credentials, servers were required to support password authentication and applications were granted unnecessarily broad access to user data. [13]

OAuth 2.0 introduced an authorization layer that decouples the client's role from that of the resource owner. Instead of credentials, the registered client receives tokens from an authorization server, which then allows for controlled access to resources stored on the resource server. During the client's registration with the authorization server, the client receives its client credentials: a client identifier and a client secret. Depending on whether the client can securely store its credentials, there are two different types of clients recognized in the OAuth protocol: public clients, which cannot store their client credentials securely and confidential clients, which can. There are four main roles in the protocol: Resource Owner (the user who owns the data), the Client (the application requesting access), the Authorization Server (the server which authenticates the resource owner and issues tokens) and the Resource Server (the host of the protected data). [13] These four roles interact with each other in the OAuth 2.0 protocol flow as follows:

1. The client requests permission from the resource owner via the authorization server.

2. The resource owner grants authorization to the client.

3. The client sends this grant to the authorization server and authenticates.
4. The authorization server validates the grant and returns an access token.
5. The client uses the token to request the protected resource from the resource server.
6. The resource server verifies the token and returns the requested data. [13]

2.3.1 Authorization Grant Types

Authorization grant serves as proof that the resource owner has granted the client permission to access its protected resources on their behalf. The client uses this proof to obtain an access token from the authorization server. OAuth 2.0 specifies four different grant types: Authorization Code, Implicit, Resource Owner Password Credentials (ROPC) and Client Credentials. [13]

Time has shown that some of the grant types are no longer considered secure. In modern client-side mobile and web applications, it is generally encouraged to use the Authorization Code grant type, since it is considered the most secure. Another widely used grant type is the Client Credentials grant, which is more of a system-level grant designed for communication between systems, such as backend services. It is considered secure, since backend services are often confidential clients that are able to keep their credentials safe. [17]

The Implicit and ROPC grant types have been removed from the latest revision of the OAuth specification (version 2.1) due to security concerns [17]. This thesis will focus primarily on the Authorization Code grant type, as it is the most secure and widely recommended approach for modern client-side applications. Figure 2.1 shows how Authorization Code flow works. It proceeds through the following steps, given that access is granted to the client:

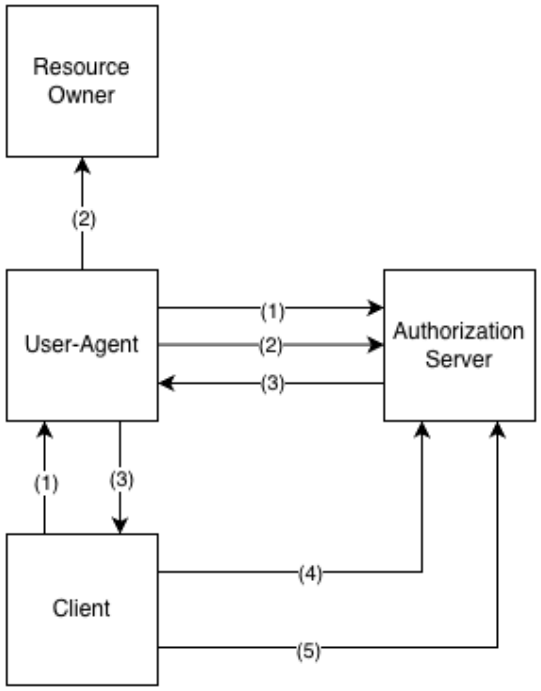

Figure 2.1: Authorization Code Flow [13]

1. The client directs the resource owner's user-agent, typically a web browser, to the authorization server's authorization endpoint. In this request, the client includes its client ID, scope and a redirect URI.

2. The user authenticates and gives consent to the client.

3. The authorization server redirects the user back to the client by using the previously given redirect URI in step (1), which now contains the authorization code.

4. The client exchanges this code for an access token by authenticating itself with the authorization server.

5. The authorization server authenticates the client, validates the authorization code and the redirect URI and responds with an access token and, optionally, a refresh token. [13]

2.3.2 Token Types

After successful authorization, the client is typically issued two types of tokens: the access token and the refresh token. Both are often represented as JSON Web Tokens (JWTs) and serve distinct purposes within the OAuth protocol. The access token functions as a kind of key that the client uses to access protected resources on behalf of the resource owner. It contains information about the allowed scopes and how long the access remains valid. [13]

Access tokens are used to make requests to servers that host protected resources. They are transmitted as bearer tokens, which are security tokens that can be used by any possessor, i.e. a bearer, in the same manner. This means that using the token does not require the bearer to prove possession of any cryptographic key material. Most commonly, access tokens are sent in the Authorization request header using the Bearer authentication scheme. [18] An example of an HTTP GET request with a bearer token is shown in Figure 2.2.

```
GET /resource HTTP/1.1
    Host: server.fi
    Authorization: Bearer ejxII6Uyhg
```

Figure 2.2: HTTP GET request with Authorization request header using the Bearer scheme

Refresh tokens have a simpler role: they allow the client to obtain new access tokens without requiring the resource owner to re-authenticate. To do this, the client authenticates itself with the authorization server and presents the refresh token, which corresponds to step (4) in Figure 2.1. Although refresh tokens are optional according to the OAuth specification, they are widely used to maintain the client's session. [13]

2.3.3 Scopes

OAuth defines scopes, which are a mechanism used to control the data and operations that a client is allowed to access. During the authorization process, the client includes one or more scopes in its access request and the authorization server issues an access token containing these scopes. It is then the responsibility of the resource server to ensure that the scope in the access token is valid. There are no specific scope values defined in OAuth itself, as they depend heavily on the system's architecture. [13], [19]

2.3.4 Proof Key for Code Exchange

Introduced in RFC 7636, Proof Key for Code Exchange (PKCE) serves as a security-enhancing extension to the Authorization Code Flow. It is used to protect public clients from authorization code interception attacks, in which an attacker could intercept the authorization code during the redirect step (step (3) in Figure 2.1). The attacker could then use the stolen code to exchange it for an access token from the token endpoint and gain unauthorized access to the resource owner's account. [20] PKCE is widely used and is considered an essential component of OAuth 2.1 [17]. It introduces an additional layer of security to the Authorization Code Flow by using the `code verifier` and `code challenge` strings:

1. The client generates a random code verifier string consisting of 43–128 characters.

2. The client applies a hashing algorithm to the code verifier to generate the code challenge string. (The specific algorithm may vary depending on the authorization server implementation, but S256 is mandatory.)

3. The client sends the code challenge along with the authorization request to the authorization server (step (1) in Figure 2.1).

4. When redeeming the authorization code for an access token, the client sends the original code verifier (step (4) in Figure 2.1).

5. The authorization server hashes the received code verifier and compares it with the previously stored code challenge. If they match, the access token is issued to the client. [20]

2.3.5 OpenID Connect

OpenID Connect (OIDC) was introduced in 2014 as an identity layer built on top of the OAuth 2.0 protocol to address its lack of built-in authentication. It allows clients to verify the identity of users after authentication and to obtain default profile information about the authenticated user. There are two additional entities defined in OIDC's specification: the OpenID Provider, which is an Authorization Server that implements OIDC and the Relying Party, which is an OAuth 2.0 client that uses OIDC. One of the main extensions introduced by OIDC is the ID token, a JWT that allows clients to verify the identity of authenticated users. The ID token is a security token that contains claims, which are pieces of information about the user's authentication. [14] The default claims included in the ID token are listed in Table 2.1 and a decoded example in JSON is illustrated in Figure 2.3.


```
{
  "iss": "https://server.fi",
  "sub": "63a01858-b47a-4817-bc90-e6b67a8f1146",
  "aud": "741624804392",
  "exp": 1761684977,
  "iat": 1761684577
}
```


Figure 2.3: Decoded ID token [14]

In OIDC, there are both standard and additional claims that a third-party application can use to retrieve basic profile information about a user [14]. In addition

Name	Required	Description
iss	required	Issuer Identifier, typically the OpenID Provider's URL
sub	required	Unique identifier for the user
aud	required	Audience(s) for which the token is intended, typically the Relying Party's client identifier
exp	required	Expiration time of the token (Unix timestamp)
iat	required	Time at which the token was issued (Unix timestamp)
auth_time	conditional	Time when the user authentication occurred (Unix timestamp)
nonce	conditional	Value used to associate the client's session with the ID token
acr	optional	Authentication Context Class Reference
amr	optional	Authentication Methods Reference
azp	optional	Authorized party, the intended recipient of the token

Table 2.1: ID token claims [14]

to the claims listed in Table 2.1, OIDC defines a set of standard claims, shown in Table 2.2. Additional claims can be defined by the OpenID Provider.

OIDC clients use scope values to define what resources an access token grants access to. The authorization request must always include the mandatory `openid` scope, which indicates that the client intends to use OIDC. Each additional scope corresponds to a specific set of claims. The standard scopes are: `profile`, `email`, `address` and `phone`. The `profile` scope includes most of the standard claims shown in Table 2.2, except for claims that are related to email, phone and address, which have their own dedicated scopes. In addition to using scopes, claims can also be requested explicitly through the claims request parameter. [14]

Besides the ID token, claims can also be retrieved from an OAuth 2.0 protected UserInfo endpoint. To access claims stored at the UserInfo endpoint, the client sends a request to the endpoint using an access token. The UserInfo request can be made using either HTTP GET or POST and the access token must be included as a Bearer token in the Authorization header. [14]

Name	Type	Description
name	string	Full name
given_name	string	Given name(s)
family_name	string	Last name(s)
middle_name	string	Middle name(s)
nickname	string	Casual name
preferred_username	string	Username
profile	string	Profile page (URL)
picture	string	Profile picture (URL)
website	string	Web page or blog (URL)
email	string	e-mail address
email_verified	boolean	true if e-mail address is verified, otherwise false
gender	string	Gender
birthdate	string	Birthday, in ISO 8601-1 format YYYY-MM-DD
zoneinfo	string	Timezone, from IANA Time Zone Database
locale	string	Locale, in BCP47, such as en_GB, fi_FI
phone_number	string	Telephone number
phone_number_verified	boolean	true if phone number is verified, otherwise false
address	JSON object	Postal address
updated_at	number	Time, when the users information was last updated

Table 2.2: Standard claims, excluding the sub claim [14]

2.3.6 OAuth 2.1

OAuth 2.1 is the latest version of the OAuth protocol. It is currently published as an Internet Draft by the IETF. OAuth 2.1 merges the lessons learned and best practices from the OAuth 2.0 family of specifications and extensions such as PKCE (RFC7636), Security Best Current Practice (RFC9700) and Browser-Based Application guidelines. [17] The goal of OAuth 2.1 is to simplify implementation, promote security and reduce the number of insecure configurations that were previously common in OAuth 2.0 deployments. More specifically OAuth 2.1:

- Deprecates the Implicit and Resource Owner Password Credentials grants due to their security considerations.
- Requires PKCE in the Authorization Code grant type.
- Prohibits the use of bearer tokens in URI query parameters.
- Requires that refresh tokens are either sender constrained or one time use for public clients. [17]

2.4 Security Context for Token Management

Security is a fundamental property of modern information systems. In information systems, security refers to the protection of systems from unauthorized access, use, disclosure, disruption, modification or destruction, in order to secure confidentiality, integrity and availability [21]. These three principles form the CIA triad, which is used to define the core security objectives. Confidentiality ensures that data is accessible only to authorized entities, integrity guarantees that information has not been modified or deleted without authorization and availability ensures that systems and data can be accessed whenever needed [22]. In the context of tokens, these principles translate directly into practical security objectives: tokens must remain secret (confidentiality), must not be tampered with or forged (integrity) and must be retrievable and usable when required (availability).

A central element of modern systems is the use of tokens. Access tokens, ID tokens and refresh tokens carry information about the entity's authentication and authorization between clients and servers [13], [14]. Tokens function as credentials and their compromise allows attackers to impersonate users or gain unauthorized access to protected resources. Therefore, the primary security concern related to tokens is their confidentiality. Integrity and availability are relevant but they are considered secondary in the context of this thesis.

The main challenge in hybrid applications is that token management must be implemented consistently across two environments with different security characteristics. Native mobile secure storage APIs, which are often accessed through hybrid frameworks' wrapper implementations, are not available in browser environments. Similarly, secure cookie mechanisms used on the web may not be replicated within mobile environments. Therefore, a consistent and secure token management strategy across both platforms requires good architectural decisions and a solid understanding of web and mobile environment security.

The Open Worldwide Application Security Project (OWASP) documents well-known security risks for both web and mobile applications. Some of these risks can affect tokens in hybrid environments. OWASP maintains separate Top 10 lists for web and mobile application security. The mobile list was updated in 2024 and the web list in 2025. Both share several common risks that come from neglecting security best practices in areas such as configuration, access control, validation, authorization, authentication and cryptographic operations. From these lists, the categories that are most relevant to token management in hybrid applications include broken access control, authentication failures, insecure design and storage, software supply chain failures, injection vulnerabilities, security misconfiguration, improper credential usage, insecure authentication and authorization and insufficient input and output validation. [23], [24] These categories guide the selection of threats, which are addressed in Chapter 3.

3 Token Storage Mechanisms and Threats

This chapter examines the security challenges related to tokens in hybrid applications. It first introduces storage mechanisms that are available on both browser-based and native mobile environments. The chapter then conducts threat modeling, which aims to answer **RQ1**: "Which threats pose the greatest risk to tokens in hybrid applications?".

3.1 Storage Options

3.1.1 Web Environment

Web Storage API

The Web Storage API provides two synchronous mechanisms for storing key–value pairs: `localStorage` and `sessionStorage`. `localStorage` is partitioned by origin, which means that all documents sharing the same origin can access the same storage instance. Additionally, its data persists across browser sessions. `sessionStorage` is also partitioned by origin, but it is scoped to individual browser tabs and its data persists as long as the browser tab is active.[25]

Cookies

Cookies are a long-established client-side storage mechanism that has historically been used to persist data across browser sessions. Before the introduction of Web Storage APIs, cookies were the primary method for maintaining client-side state in web applications. Cookies are small, with a typical maximum size of around 4 KB. A cookie consists of a key–value pair stored by the browser and is automatically included in requests to the server. [26], [27] They can be configured with several attributes that determine their lifetime, scope and security properties. These attributes are listed in Table 3.1.

IndexedDB

IndexedDB is an asynchronous, browser-based database that allows web applications to store data on the client-side. Unlike the previously mentioned simpler key-value storage mechanisms, IndexedDB can store not only basic values like integers or strings, but also complex objects, such as images, videos and other binary data. It is a NoSQL, object-oriented database designed around JavaScript objects. IndexedDB offers significantly greater storage capacity compared to other client-side storage mechanisms in the browser. It also supports applications that function offline by persisting data locally. IndexedDB can be used through the IndexedDB interfaces, such as `IDBTransaction` and `IDBRequest`. [28]

In-memory

In addition to platform-specific mechanisms, there is a storage option that applies across both web and mobile environments: storing data in the application's memory, for example, as variables. However, the data stored in-memory is short-lived, since when the page is refreshed or the application restarts the stored data is lost.

Attribute	Values	Description
Expires	UTC date string	Sets an expiration date for the cookie. After this date, the browser deletes it.
Max-Age	Integer (seconds)	Sets the lifetime of the cookie relative to the current time. If both `Expires` and `Max-Age` are set, `Max-Age` takes precedence.
Domain	Domain string, e.g. `example.fi`	Defines the host(s) the cookie is sent to. If specified, subdomains are included. If omitted, the cookie applies only to the exact host that set it.
Path	Path string, e.g. `/`, `/example`	Restricts the cookie to requests whose path matches the given value (including its subpaths).
Secure	Flag	Ensures the cookie is only sent over secure connections (HTTPS).
HttpOnly	Flag	Prevents client-side scripts (e.g. JavaScript) from accessing the cookie.
SameSite	`Strict`, `Lax`, or `None`	Controls whether the cookie is sent on cross-site requests. `Strict`: sent only on same-site requests. `Lax`: sent on same-site requests and on top-level navigation. `Lax` is the default.

Table 3.1: Cookie attributes [26]

3.1.2 Mobile Environment

This subsection examines the storage mechanisms provided by the Expo framework, since it is the primary framework used in this thesis as noted in Section 2.1.

SecureStore

In hybrid mobile applications, native secure storage mechanisms are typically accessed through wrapper libraries, such as Expo's Secure Store or React Native's

Encrypted Storage, which provide a unified API for both iOS and Android. On Android, Expo's SecureStore stores values in `SharedPreferences` and encrypts them with the Keystore system. On iOS, SecureStore stores values in the Keychain under the `kSecClassGenericPassword` class. [29]

The iOS Keychain is designed to store small but sensitive data. It provides a secure storage system that encrypts each item before storing it. Keychain entries are encrypted with two keys: a metadata key and a per-row secret key. The metadata key protects the Keychain's metadata and the per-row key protects the actual secret value. Keychain data is stored internally as a single SQLite database, with the `securityd`-daemon enforcing access controls. Applications can access only those Keychain items for which they have the appropriate access rights, such as the correct application identifier or access group. [30]

The Android Keystore system provides a similar mechanism for securely storing cryptographic keys. Keys are generated and stored in a hardware-backed environment, typically a Trusted Execution Environment (TEE) or Secure Element and are marked as non-exportable. This means the raw key material cannot be extracted, even by the application itself. Instead, applications can request cryptographic operations such as signing or encryption to be performed inside the secure environment. The result of the operation is returned while the key never leaves the Keystore. [31]

FileSystem and SQLite

The Expo framework provides the FileSystem and SQLite modules for data storage. SQLite offers a lightweight relational database system that can be accessed through the SQLite API. [32] The FileSystem module gives access to the device's local file system and supports common file operations such as reading, writing, moving and copying files and directories. [33]

AsyncStorage

AsyncStorage is a commonly used global, unencrypted key-value storage system originally built for React Native. It is inspired by the Web Storage APIs used in browsers, but unlike them it works asynchronously. On Android, AsyncStorage relies on SQLite, while on iOS, values shorter than 1024 characters are stored in the `manifest.json` file and larger values are placed in individual files. In browser-based environments, values are stored in the browser's `localStorage`. [34] In AsyncStorage's upcoming iteration, version `3.0-next`, mobile platforms will use SQLite under the hood and web browsers will use IndexedDB [35]. This thesis examines version 2.0.

Other libraries

A wide variety of community-maintained libraries exist for storing data in React Native, these offer different trade-offs in performance, platform support and security. Some provide fast, synchronous key-value storage optimized for frequent reads and writes, while others use asynchronous APIs suitable for storing simpler or larger amounts of data. Certain libraries include encryption for sensitive data, while others prioritize ease of use over security. Maintenance levels and quality vary, so it is important to consider factors such as cross-platform compatibility, performance requirements and security guarantees when selecting a specific option. [36] These options are not considered further in this thesis.

3.2 Threat Modeling

Threat modeling is a process used to identify and understand potential threats and mitigations for systems and their valuable assets. It provides an organized view of a system and its environment from a security perspective. [37] In this thesis, threat

modeling is used to evaluate which threats pose the greatest risk to tokens in hybrid applications. To support this analysis, the STRIDE model is used to categorize the threats at a high level. STRIDE is a common threat modeling framework originally introduced by Microsoft, which groups security threats into six categories: Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service and Elevation of Privilege. [38] The analysis is based on a review of OAuth security guidance and documented vulnerabilities, including RFC 6819 [39], RFC 9700 [40], OAuth 2.0 for Browser-Based Applications [41], OAuth 2.0 for Native Apps [42] and the OWASP Top 10 lists [23], [24]. From these sources, relevant threats are selected, assessed by likelihood and impact, and categorized using STRIDE.

3.2.1 Evaluation Criteria

Each threat is evaluated for its likelihood and impact as shown in Tables 3.2 and 3.3. This evaluation is performed alongside the STRIDE categorization in order to determine the greatest risk to tokens in hybrid applications, which is derived from the combination of likelihood and impact.

Likelihood is evaluated based on how vulnerable the system's components generally are and what the preconditions for an attack are. Threats that target components that are generally more exposed are considered more likely, while threats that require multiple specific preconditions or additional capabilities on the part of the attacker are considered less likely. Impact is evaluated by the potential consequences of a successful attack. Scenarios which lead to a full compromise of tokens or user accounts are rated as high impact and those causing limited or temporary effects are rated lower.

Score	Definition
High	The threat exploits commonly exposed components with no significant preconditions
Medium	The threat requires specific preconditions, such as user interaction or misconfiguration
Low	The threat requires multiple preconditions, such as physical access and bypassing platform protections

Table 3.2: Likelihood evaluation

Score	Definition
High	Severe consequences, such as full compromise of tokens or user accounts
Medium	Moderate consequences, indirect compromise, such as client hijacking
Low	Minimal consequences, such as minor inconvenience or temporary disruption

Table 3.3: Impact evaluation

3.2.2 Threat Analysis

The threats selected for this analysis include authorization code interception, cross-site scripting (XSS), insecure storage, phishing, device theft, man-in-the-middle (MITM) attacks, malware infections, software supply chain attacks, leakage, cross-site request forgery (CSRF) and insufficient token lifecycle management.

Authorization Code Interception

Authorization code interception is an attack in which an attacker obtains the authorization code during the redirect step of the Authorization Code Flow and then exchanges it for tokens. This threat is particularly relevant to hybrid applications, since authorization responses are delivered through redirect mechanisms that differ between platforms. In browser-based environments, authorization codes may be exposed through referrer headers, browser history or logs. In mobile environments, a malicious application installed on the same device may register the same custom URI scheme as the legitimate client and intercept the authorization response. [20],

[40], [42] The likelihood of authorization code interception is considered medium. The attack requires specific preconditions, such as malicious application registered for the same redirect URI scheme on mobile environments or insufficient redirect URI matching on the authorization server. However, PKCE effectively mitigates this threat by binding the authorization code to the original client using code verifier[20], [40]. The impact is considered high, since an attacker who successfully exchanges the authorization code can obtain valid access and refresh token, which allow for unauthorized access. As a result, the overall risk is rated as medium to high. This threat aligns primarily with the STRIDE categories of Spoofing and Information Disclosure.

Cross-Site Scripting

Cross-site scripting (XSS) is a well known vulnerability in browser-based environments that allows attackers to inject malicious scripts into an application. These scripts can access sensitive information stored in JavaScript accessible mechanisms such as variables, storage and perform actions on behalf of the authenticated user. [43], [44] XSS is particularly relevant threat to applications that store tokens in client-side storage, since the tokens are directly readable by any script. The likelihood of XSS is considered high. Input fields and other user controlled inputs are typically exposed in web applications and a successful attack only requires that one of these components is vulnerable. Furthermore, injection vulnerabilities are one of the most prevalent categories of web application weaknesses [45]. The impact is also considered high. A successful XSS attack can expose tokens, which allows for impersonation of the user. Therefore, the overall risk is considered high and in the STRIDE model XSS primarily relates to Information Disclosure and Tampering.

Insecure Storage

Insecure storage refers to persisting tokens in locations that are accessible to unauthorized parties. In hybrid applications tokens can be placed in mechanisms such as `localStorage`, `sessionStorage`, IndexedDB, SQLite databases or the filesystem. These storage options may be readable by scripts running within the application or by other malicious applications. [39], [41], [46] The likelihood of insecure storage is considered high, since hybrid application commonly rely on general-purpose storage mechanisms that are accessible to client-side code by default. Several widely used options, such as AsyncStorage on mobile and the Web Storage APIs in browsers persist tokens in plain text without built-in protection [25], [34]. Platform-specific secure storage or additional infrastructure is required to be able to achieve stronger confidentiality guarantees. The impact is also considered high, since insecure storage exposes tokens passively. If an attacker gains the required access, the tokens can be read and used to impersonate the user. The overall risk is therefore considered high and in STRIDE this threat primarily relates to Information Disclosure.

Phishing

Phishing is an attack in which an attacker attempts to trick a legitimate user into interacting with a malicious service in order to obtain tokens or other sensitive information. An attacker might impersonate a resource server and trick a client into sending it an access token issued by a legitimate authorization server. Once the malicious resource server obtains this token, it could use it to gain unauthorized access to other services. There are additional phishing attacks that mainly target tokens, such as an attack in which a malicious party impersonates an authorization server to obtain refresh tokens. The likelihood of a phishing attack is considered medium, since even though social engineering attacks are widespread and do not require technical vulnerabilities in the application, they can be mitigated with fol-

lowing basic best practices, such as not sending access tokens to unknown servers, restricting tokens to a certain resource server and more. [39] However, if tokens or credentials are successfully obtained, attackers may impersonate legitimate users and gain unauthorized access to protected resources. Therefore, the impact of phishing is considered high. As a result, the overall risk is medium to high and from the STRIDE perspective, phishing primarily relates to the Spoofing and Information Disclosure categories.

Device Theft

Device theft occurs when an attacker gains physical access to a user's device after it is lost or stolen. If tokens are persisted in client-side storage without proper protection, the attacker may retrieve them directly from the device. [47] The likelihood of device theft is considered low. Successful token extraction requires physical access to the device and the capability to bypass platform-level protections such as device locks, encryption or hardware-backed secure storage. The impact is considered high. Stored tokens grant access to the legitimate user's data. Device theft may additionally expose other credentials stored on the device. Revoking tokens and invalidating sessions is particularly important in this context, since the exposure window may be longer, due to the fact that device loss may not be noticed immediately. The overall risk is therefore medium and in STRIDE this threat relates to Spoofing and Information Disclosure.

Man-in-the-Middle Attacks

Man-in-the-middle (MITM) attacks occur when an attacker intercepts communication between two parties and is able to observe or modify transmitted data. Attackers may intercept tokens, authorization codes or client credentials during transmission if transport layer security (TLS) is absent or improperly configured [39].

Furthermore, misconfigurations such as mixing TLS and non-TLS content, insecure endpoints or inappropriate certificate validation can increase the risk of MITM attacks [48]. The likelihood of MITM attacks is considered medium, since modern applications typically enforce TLS, which reduces the likelihood of this attack. However, specific preconditions, such as misconfigurations may be present in real-world deployments, especially in hybrid application where both browser and mobile environment must be configured correctly. The impact is considered high, since intercepted tokens allow attackers to impersonate legitimate users. Additionally, MITM attack may remain undetected. The overall risk is therefore medium to high, and in STRIDE MITM attacks relate to Information Disclosure and Tampering.

Malware Infections

Malware infections compromise the security of client devices and may expose sensitive information in applications. In browser-based environments, malicious software may monitor browser profiles or extract data from locally stored files. On mobile platforms, a related concern is malicious applications installed on the same device, which could use weak file permissions to read data belonging to other applications. Additionally, malware may further exploit rooted or jailbroken devices to bypass operating system level security controls and encryption mechanisms. [41], [42], [46] In these cases, tokens stored within the application environment become accessible to attackers. The likelihood of malware infection is considered medium, since a successful compromise typically requires specific preconditions, such as the user installing a malicious application or visiting a compromised resource. The impact is considered high, since malware may persist on the device and continuously monitor application behavior, which allows it to capture tokens as they are stored or used. The overall risk is therefore rated as medium to high and in STRIDE this primarily relates to Information Disclosure.

Software Supply Chain Attacks

Software supply chain attacks happen when malicious or compromised third-party dependencies introduce vulnerabilities into an application. Modern applications rely heavily on external libraries and frameworks, which may become attack vectors if they are compromised. A malicious dependency may intercept authentication flows, capture tokens or introduce other vulnerabilities into the application. [49] The likelihood of supply chain attacks is considered medium. Modern hybrid application commonly depend on a large number of third-party libraries, which expands the attack surface. However, successful exploitation requires the compromise of a dependency into production systems without detection. The impact is considered high, since a compromised dependency may access tokens during the authentication flow, at rest or while in use. Furthermore, the compromised dependency may extend to broader system compromise, depending on its capabilities. The overall risk is thereby considered medium to high. In STRIDE this threat primarily aligns with Tampering and Information Disclosure.

Leakage

Leakage refers to tokens being exposed through unintentional means rather than through a targeted attack. Tokens may be leaked through client or server logs, errors or crash reports, browser history, URL parameters or through unencrypted storage options, that can be read by anyone with access to the device or application data. [39], [41], [46], [50] The likelihood of leakage is considered medium, since modern browsers and protocols provide protections, such as the use of authorization headers instead of URL parameters for transmitting tokens. However, tokens may still be exposed through multiple system components, such as logging mechanisms or application configuration if the application includes insecure design patterns. The impact of token leakage is high, since sensitive tokens could be accessed by

unauthorized parties and then be used to access protected resources or impersonate legitimate users. The overall risk is considered medium to high due to the severity of potential consequences. Regarding STRIDE categories, token leakage falls primarily under Information Disclosure.

Cross-Site Request Forgery (CSRF)

Cross-site request forgery (CSRF) is an attack in which an authenticated user unintentionally executes unwanted actions on a target application. CSRF attacks primarily exploit cookie-based authentication by sending forged requests from a malicious web page to a trusted application, which causes the request to be processed using the legitimate user's existing session. [27], [51] This threat is primarily relevant in web browsers. However, a similar attack may occur on mobile platforms called cross-app request forgery, where malicious applications abuse inter-app communication during the authorization request [42]. CSRF is evaluated as medium likelihood. Although CSRF is a well-known web vulnerability, it requires specific preconditions, such as the use of cookie-based authentication and lack of request validation mechanisms. Protections, such as the `SameSite` cookie attribute and the use of CSRF tokens, reduce the possibility of successful attacks [26]. Furthermore in OAuth authorization flows, the `state` parameter is recommended to mitigate CSRF attacks [42]. The impact of CSRF is considered medium. While attackers may perform unauthorized actions on behalf of a legitimate user, CSRF does not directly expose tokens and is limited to actions permitted by the user's existing session [51]. As a result, the overall risk is considered medium and it aligns with the Spoofing category in STRIDE.

Insufficient Token Lifecycle Management

Insufficient token lifecycle management is a security concern that affects how long compromised tokens remain usable. Practices such as issuing long-lived access tokens, not rotating refresh tokens, not sender-constraining tokens and not using token revocation mechanisms increase the time window in which compromised tokens remain valid. These amplify the potential impact of token compromise. Refresh token rotation is particularly important in public clients, which need to rely only on rotation and revocation [39], [40], [41], [52]. The likelihood of insufficient token lifecycle management is considered medium, since it requires specific preconditions such as misconfigured token lifetimes or inconsistent rotation policies. The impact is assessed as medium. Insufficient token lifecycle management does not directly expose tokens, but it significantly increases the time window during which compromised tokens remain valid. As a result, the overall risk is rated as medium. This threat aligns primarily with the STRIDE category of Information Disclosure.

Threat	**STRIDE**	**Likelihood**	**Impact**	**Risk**
Authorization code interception	S, I	Medium	High	Medium–High
XSS	T, I	High	High	High
Insecure Storage	I	High	High	High
Phishing	S, I	Medium	High	Medium–High
Device Theft	S, I	Low	High	Medium
MITM	T, I	Medium	High	Medium–High
Malware Infections	I	Medium	High	Medium–High
Supply Chain Attacks	T, I	Medium	High	Medium–High
Leakage	I	Medium	High	Medium–High
CSRF	S	Medium	Medium	Medium
Insufficient Token Lifecycle Management	I	Medium	Medium	Medium

Table 3.4: Threat assessment for tokens in hybrid applications

3.2.3 Discussion

The threat analysis shows that tokens in hybrid applications are exposed to a wide range of threats. None of the threats fall into the low-risk category, which indicates that all of the selected threats are relevant to tokens in hybrid applications. These threats may originate from the application environment, the client device or the OAuth configuration. The identified threats can be broadly separated into three different categories: threats that directly compromise tokens, threats that exploit protocol or configuration weaknesses and environmental and threats that target the environment or infrastructure. Table 3.4 summarizes the results. In answer to **RQ1**, XSS and insecure storage pose the greatest risk to tokens in hybrid applications, since both are assessed as high risk due to their combination of high likelihood and high impact.

The first category includes XSS, insecure storage, leakage, device theft and phishing. These threats represent token compromise directly, since they allow attackers to obtain tokens without compromising another components. XSS and insecure storage are evaluated as high risk due to their high likelihood and high impact. Leakage and phishing are evaluated as medium to high risk. Device theft is evaluated as medium risk due to its low likelihood.

The second category consists of protocol and configuration related threats such as authorization code interception, insufficient token lifecycle management and CSRF. These threats typically do not expose tokens directly but weaken the overall security of the system. For example, long-lived tokens or the absence of refresh token rotation may significantly increase the impact of other threats by extending the time window in which compromised tokens remain valid.

The third category includes threats that target the underlying environment or infrastructure, such as MITM attacks, malware infections and software supply chain attacks. These threats affect the broader system rather than the application logic

itself. The likelihood is generally lower due to required preconditions, but their impact remains high since successful exploitation may expose tokens and other sensitive data.

From the STRIDE perspective, the analysis shows that most of the threats fall into the Information Disclosure and Spoofing categories. This is because the tokens serve as credentials, which means that if they are compromised, attackers can impersonate legitimate users and gain access to protected resources. Tampering also appears in threats that involve active modification of application behavior or data, such as XSS, MITM and supply chain attacks. In contrast, threats that are related to Repudiation and Denial of Service are not as relevant in the context of client-side token management.

Hybrid applications operate in both browser-based and native mobile environments, which introduces additional complexity to the threat landscape. Many of the identified threats are not limited to a single environment. For example, XSS and CSRF primarily affect browser environments, while device theft can be more prevalent in the mobile environments. The other threats: authorization code interception, insufficient token lifecycle management, insecure storage, malware infections, leakage, phishing, MITM and supply chain attacks are relevant in both environments. Overall, the primary security concern associated with tokens in hybrid applications relates to their potential exposure rather than availability. Protecting tokens requires a comprehensive approach that includes secure client-side storage, strict transport layer security, correct OAuth configuration and careful token lifecycle management in both browser-based and native mobile environments.

4 Evaluation of Token Storage Mechanisms

The goal of this chapter is to answer **RQ2**: "How do different token storage mechanisms compare in terms of confidentiality and implementation cost?". The evaluation is conducted by integrating each mechanism into a hybrid application, followed by a qualitative assessment based on the criteria defined in Section 4.1.

4.1 Evaluation Criteria

4.1.1 Confidentiality

The confidentiality evaluation examines how effectively each storage mechanism protects tokens from unauthorized access. The assessment focuses on the properties of the storage mechanism: how accessible stored tokens are at runtime, does the mechanism provide isolation or encryption by default and how long tokens remain exposed in the storage. These properties determine how well each mechanism withstands the storage-level threats identified in Section 3.2, such as XSS in the browser environment and insecure storage practices on both platforms.

Confidentiality is primarily assessed for tokens at rest, since some degree of runtime exposure is unavoidable for any storage mechanism. Other threats can be mentioned, if they relate to the storage option in a meaningful way. Confidentiality

is scored using a three-point ordinal scale (low, medium, high). Table 4.1 provides a summary of the interpretation for each level.

Criteria	Low	Medium	High
Confidentiality	Tokens persist in locations that are accessible at runtime and have no built-in protection	Tokens are accessible at runtime, but the exposure window is significantly reduced	Tokens are protected by default through encryption or by being isolated from runtime access

Table 4.1: Interpretation of low, medium and high confidentiality

Storage mechanisms that limit token accessibility, reduce the exposure window, provide isolated environments or apply encryption are considered to provide stronger confidentiality. In contrast, mechanisms that store tokens in easily accessible locations without built-in protections provide weaker confidentiality.

4.1.2 Implementation Cost

The implementation cost evaluation assesses how seamlessly each storage mechanism can be integrated into a hybrid application. In this thesis, implementation cost refers to the needed development effort and complexity related to the storage mechanism. The following factors are taken into account when estimating implementation cost:

- API complexity: The simplicity of the core storage functions such as storing, retrieving and clearing tokens without unnecessary boilerplate.
- Configuration overhead: The additional work required to implement the storage mechanism, such as the need for external dependencies, initialization or additional infrastructure.
- Cross-platform implementation effort: The extent to which the same storage

mechanism and its implementation can be reused across web and mobile environments

Each criterion is scored on a three-point ordinal scale (low, medium, high) based on the level of development effort required. Table 4.2 defines how each level is interpreted for each criterion.

Criteria	Low	Medium	High
API complexity	Direct use of storage APIs with minimal boilerplate	Requires wrapper utilities or moderate exception handling	Requires custom implementation
Configuration overhead	Requires no additional setup	Requires moderate initialization or configuration	Requires additional infrastructure or extensive configuration
Cross-platform implementation effort	Same implementation works consistently across web and mobile platforms	Equivalent functionality can be achieved with limited platform-specific logic or alternative storage mechanisms	Equivalent functionality requires separate implementations or architectural difference across platforms

Table 4.2: Interpretation of low, medium and high implementation cost

Storage mechanisms with mostly low values across the criteria are considered to have low implementation cost as they require little to no additional configuration and behave consistently across environments. In contrast, mechanisms with mostly high values are considered to have high implementation cost, typically due to platform-specific implementations or extra initialization logic.

4.2 Storage Mechanism Evaluation

4.2.1 Implementation

Each storage mechanism was evaluated by using a hybrid application built with the Expo framework [2] (SDK v54). The application supports both web and mobile environments. Keycloak [53] (v26.4.7) was used as the identity provider for authorization and authentication. A Quarkus [54] (v3.20.5) backend was used to expose protected API endpoints and, if applicable, it was used to manage server-side session.

For all storage mechanisms except cookies, the Expo client application communicates directly with Keycloak using the OAuth 2.0 Authorization Code Flow with PKCE and stores the tokens. The cookie-based authentication is implemented by a Quarkus backend, which works as a confidential OAuth client and maintains the authenticated session using `HTTPOnly` cookies. The Quarkus backend also uses Authorization Code Flow with PKCE enabled. Each storage mechanism was integrated into the same application through a common storage interface. The storage mechanism was selected via the application's configuration variables. The implementation can be found in a repository on GitHub: `https://github.com/Uteeaami/token-storage-evaluation`.

4.2.2 In-Memory

Confidentiality

In-memory storage limits the token's exposure window to the application's active lifetime. Tokens are discarded once the application is closed or the page is refreshed, which reduces their accessibility. In-memory storage's security can be increased by using practices such as closures, in which the token is only accessed through predefined functions. However, tokens can still be extracted if the application is compromised. For example, if an attacker uses prototype poisoning to replace the

functions used by the closures and intercept the tokens. [41]

In browser environments, the main concern for tokens stored in-memory is that they are prone to XSS vulnerabilities. If an XSS vulnerability exists, tokens held in memory can be read by injected scripts. Common mitigation strategies for XSS include output encoding and sanitization and the use of strict Content Security Policies (CSP). [43] However, these do not eliminate the inherent problem.

On mobile platforms, the token exposure window is greater due to how the application lifecycle behaves. Mobile operating systems do not necessarily terminate applications when users leave them. Applications may be pushed to the background, where they enter a stopped state but remain in memory. During this state, in-memory tokens persist until the operating system evicts the background process. [55], [56] This behavior increases the exposure window but it can be mitigated by using background refresh policies [57]. In addition, mobile operating systems benefit from an application-level sandbox, which isolates the application's resources from other applications [58], [59].

In-memory storage does not provide any built-in security mechanisms. Encrypting tokens in memory could reduce token accessibility and increase confidentiality, but this approach introduces a secondary key management problem. Secure handling of encryption keys would require hardware-backed secure enclaves or external trusted components, such as a server-side key store, which increases complexity.

In-memory storage's confidentiality cannot be rated high because it lacks built-in protection mechanisms and tokens remain accessible at runtime. At the same time, it cannot be rated low because it limits token lifetime to the application session, which reduces exposure time. Therefore, in-memory storage represents a middle ground between exposure and protection and its confidentiality is rated as medium.

Implementation Cost

In-memory token storage does not rely on a specific API and it can be implemented by using standard language functionalities such as variables, state management systems or other data structures. In this evaluation, tokens were stored using a simple key-value map to support multiple token types. No additional configuration or platform-specific setup was needed. Another significant benefit of in-memory storage is its platform independence. The same implementation can be reused in both browser and mobile environments without needing conditional logic. As a result, in-memory storage's API complexity, configuration overhead and cross-platform implementation effort are all evaluated as low, which corresponds to low implementation cost.

4.2.3 Web Storage APIs

Confidentiality

The storage mechanisms provided by the Web Storage API have different availability windows. Tokens stored in `sessionStorage` are bound to the lifetime of a single browser tab and are removed when the tab is closed, while `localStorage` persists across browser restarts [25]. This means that tokens stored in `sessionStorage` have a shorter exposure window compared to those stored in `localStorage`. However, both remain exposed for the full duration of the session.

Tokens stored that are stored in Web Storage APIs are accessible to any client-side scripts, which execute in the same origin. If an XSS vulnerability exists, stored tokens can be read by injected scripts. Both storage mechanisms are scoped to the browser's origin, which prevents scripts from other origins from accessing stored values. However, this restriction does not protect against malicious scripts executed in the same origin. The Web Storage API does not provide other built-in security

related features. Encryption could be applied, but this would introduce a secondary key management problem and significantly increase the implementation cost. Common mitigation strategies against XSS include output encoding and sanitization and the use of strict CSP. [43], [44] Since tokens stored by using the Web Storage APIs persist in locations accessible at runtime with no built-in protection and no significant reduction in exposure window, their confidentiality is considered low.

Implementation Cost

The Web Storage API provides an intuitive interface for storing key-value pairs. The interface includes functions such as `setItem`, `getItem` and `removeItem`. It requires no additional initialization, which makes it straightforward to implement. Furthermore, the Web Storage API is available in all modern browsers and does not require additional configuration or dependencies. Therefore, API complexity and configuration overhead are considered low.

The Web Storage API is only available in browser-based environments and it cannot be used in native mobile applications. If a hybrid application uses one of these mechanisms in the browser environment, it would require a separate storage implementation for mobile platforms. However, the different implementation on mobile can be implemented through a common storage abstraction or interface, which requires only a limited amount of conditional logic. Therefore, cross-platform implementation effort is evaluated as medium. The overall implementation cost of the Web Storage API is evaluated as low, since low API complexity and configuration overhead outweigh the medium cross-platform implementation effort.

4.2.4 IndexedDB

Confidentiality

IndexedDB is an asynchronous, origin-scoped database that does not provide isolation beyond the same-origin policy. As a result, tokens stored in IndexedDB are accessible to any client-side script that executes in the same origin. If an XSS vulnerability exists, stored tokens can be read by any injected scripts. Similarly to the mechanisms provided by the Web Storage API, origin-scoping does not protect against malicious scripts executed in the same origin. Common mitigation strategies against XSS include output encoding, sanitization and the use of strict CSP. [28], [43]

IndexedDB provides an asynchronous and transactional access model, which can be more complex to implement correctly when compared to simpler key-value storage mechanisms. This complexity may increase the likelihood of implementation errors, such as inconsistent cleanup or unintended data persistence, which can extend the exposure window beyond what is intended. IndexedDB does not provide other built-in security mechanisms beyond the same-origin principle [28]. Encryption could be implemented, but this would introduce a secondary key management problem and significantly increase implementation cost. Therefore, IndexedDB's confidentiality is evaluated as low.

Implementation Cost

IndexedDB introduces a more complex API than simpler key-value storage mechanisms. Its use requires database initialization and transactional access patterns. All operations are asynchronous and require explicit error handling. In the reference implementation, tokens are stored as objects which contain metadata about the token type and value.

IndexedDB supports flexible data modeling, but it requires additional boilerplate

and increases the likelihood of implementation errors when compared to simpler key-value storage options. This means that IndexedDB's API complexity is evaluated as medium. Similarly, configuration overhead is evaluated as medium, since database initialization is required, but no additional infrastructure is needed.

IndexedDB is not supported in native mobile environments, which means that hybrid applications require separate storage implementations for each platform. However, the conditional logic needed to separate between IndexedDB on the web and an equivalent mechanism on mobile is minimal. Therefore, cross-platform implementation effort is evaluated as medium. Overall, IndexedDB's implementation cost is considered medium.

4.2.5 Cookies

Confidentiality

Cookie-based storage differs from other evaluated storage mechanisms, since tokens are not accessible client-side, when the `HttpOnly` flag is used. This flag prevents JavaScript from reading the cookie, which mitigates the risk of theft through XSS. However, XSS can still be exploited to trigger authenticated requests on behalf of the user, which means that the risk of XSS must not be overlooked. [26], [43]

CSRF is a particularly relevant threat for cookie-based storage, since cookies are automatically included in outgoing requests. The risk can be reduced by configuring cookies with the `SameSite` attribute set to `Strict` or `Lax`, which limits when cookies are included in cross-site requests. [26] Additional server-side measures, such as origin or referrer validation and the use of CSRF tokens can be used to further reduce the risk [60].

Cookies provide additional transport-level protection through the `Secure` attribute, which ensures that cookies are only transmitted over HTTPS connections [26]. This attribute guarantees that cookies are not sent over unencrypted channels.

While cookies remain vulnerable to CSRF, this threat does not directly compromise tokens but rather abuses authenticated requests. However, CSRF can be mitigated by using `SameSite` and CSRF tokens. Tokens stored in `HttpOnly` cookies are protected by default through isolation from client-side code and simple mitigations can be used to mitigate the risk of CSRF. [26] Therefore, cookies' confidentiality is rated high.

Implementation Cost

Cookie-based storage introduces significantly higher complexity than client-side storage mechanisms. Since `HttpOnly` cookies cannot be accessed or managed by the client application, a backend component is required. In the reference implementation, cookie-based storage is implemented by a Quarkus backend using its built-in OpenID Connect support. Modern frameworks do provide abstraction for handling OAuth flows, session management and PKCE, but the overall setup requires additional infrastructure and configuration.

Cookies are common in browser environments, but not in native mobile applications. They can be implemented through WebView components in native applications, but this approach is outside the scope of this thesis. Additionally, React Native does not support the use of cookies by default. Therefore, cookies are considered only for browser environments, which increases the implementation cost because they are not compatible across both environments.

When implementing cookie-based storage in the web environment, the application must handle API requests differently across environments. This requires further conditional logic in a hybrid application to support cookie-based authentication on the web and bearer token authentication on native platforms.

Overall, cookie-based storage has high configuration overhead, since it requires a separate backend component to manage OAuth flows and session state. API com-

plexity is low, since cookie handling is managed by the browser through HTTP headers. Cross-platform implementation effort is rated high, since cookie-based authentication does not provide a consistent implementation across web and mobile platforms and requires platform-specific handling. As a result, the overall implementation cost is evaluated as high, since the high configuration overhead and cross-platform implementation effort outweigh the low API complexity.

4.2.6 AsyncStorage

Confidentiality

AsyncStorage provides a cross-platform key-value interface, but its underlying storage depends on the environment. In browser environments, AsyncStorage uses `localStorage`, which means that tokens are accessible to client-side scripts. On mobile platforms, data is stored in an SQLite database and on iOS in `manifest.json` or individual files depending on the size of the data. [34] On mobile, the application sandbox isolates data from other applications, which provides basic protection at the operating system level [58], [59].

AsyncStorage does not provide built-in mitigations, such as encryption [34]. Stronger protection of tokens would require its own encryption implementation, which introduces secondary key management challenges and increases implementation cost. As a result, stored tokens remain in plaintext in the underlying storage on all platforms. Since tokens persist in locations accessible at runtime with no built-in protection, AsyncStorage's confidentiality is evaluated as low.

Implementation Cost

AsyncStorage provides a simple API consisting of methods such as `setItem`, `getItem` and `removeItem`, which results in low API complexity. AsyncStorage is asynchronous, which may introduce race conditions when tokens are accessed from mul-

tiple components, such as during application initialization. These timing issues may require additional logic to synchronize the state, but the overall impact is limited. AsyncStorage requires only the installation of one dependency and its implementation can be used on both web and mobile platforms. Therefore, configuration overhead and cross-platform implementation effort are rated low. Overall, the implementation cost of AsyncStorage is evaluated as low.

4.2.7 SecureStore

Confidentiality

SecureStore provides strong confidentiality on mobile platforms by using operating system level secure storage mechanisms. On iOS, values are stored in the Keychain, where each item is encrypted before being stored. Access to the Keychain is further restricted by an application identifier or an access group. On Android, SecureStore stores values in SharedPreferences but encrypts them using keys stored in the Android Keystore, which are kept in a hardware-backed environment such as a Trusted Execution Environment. [29], [30], [31] These mechanisms are specifically designed to be used when storing sensitive data, since they provide encryption and application-level sandboxing [29], [58], [59]. As a result, tokens stored using SecureStore are significantly more resistant to unauthorized access when compared to tokens stored in unprotected storage mechanisms. Therefore, SecureStore's confidentiality is rated as high.

Implementation Cost

SecureStore provides a simple API to store encrypted key-value pairs. It includes synchronous and asynchronous functions such as `setItem` or `setItemAsync`, `getItem` or `getItemAsync` and `deleteItemAsync`. As a result, API complexity is evaluated as low. Configuration overhead is also considered low, since its usage requires only

one dependency to be installed and there is no need for extra initialization or configuration.

The downside of SecureStore is that it is limited to the mobile environment. There is no corresponding solution for browser-based environments. To be able to provide comparable confidentiality guarantees on the web, SecureStore needs to be substituted with a fundamentally different mechanism, such as `HttpOnly` cookies. This introduces an architectural difference, which leads to a high cross-platform implementation effort. Overall, SecureStore's implementation cost is evaluated as medium.

4.2.8 SQLite

Confidentiality

SQLite provides a persistent storage mechanism based on a relational database. On mobile platforms, the database file is located in the application's sandboxed environment, which protects the file from other applications [58], [59]. SQLite does not provide encryption or other confidentiality guarantees by default [32]. Encryption could be implemented, but this introduces a secondary key management problem.

In addition, SQLite requires managing the database schema and query logic manually, which may introduce insecure patterns such as inconsistent cleanup or unintended persistence. These patterns can further extend the exposure window of tokens. Since tokens remain exposed in plaintext storage and SQLite does not provide built-in protection mechanisms, its confidentiality is evaluated as low.

Implementation Cost

SQLite requires the database to be initialized with a schema and the logic for queries must be implemented manually. Compared to other key-value based storage mechanisms, this introduces additional boilerplate and complexity. As a result, API

complexity and configuration overhead are evaluated as medium.

SQLite is intended to support both mobile and web environments. However, at the time of this evaluation, the SQLite module for web environments was in an experimental state and could not be used reliably. Therefore, to be able to achieve equivalent functionality across web and mobile platforms requires substituting the storage mechanism on the web with an alternative, such as IndexedDB or the Web Storage APIs. This results in a medium cross-platform implementation effort. Overall, the implementation cost of SQLite is evaluated as medium.

4.2.9 FileSystem

Confidentiality

The FileSystem API allows applications to persist data by writing directly to files. Tokens can be stored in document or cache directories. In mobile environments, these directories are protected by the application's own sandbox [58], [59]. However, FileSystem does not provide encryption or additional security guarantees by default [33]. Encryption could be implemented, but this would introduce further key management problem and increase the overall implementation cost.

FileSystem provides basic file operations, such as creation and deletion. This provides flexibility, but also increases the risk of insecure patterns. For example, inconsistent cleanup may persist tokens longer than necessary. Since tokens remain exposed in plaintext storage and FileSystem does not provide built-in protection mechanisms for sensitive data, its confidentiality is evaluated as low.

Implementation Cost

The FileSystem API exposes simple operations for writing and retrieving data and requires the installation of one dependency. Storing tokens requires serialization logic and manual handling of file creation and cleanup. In the reference implemen-

tation, tokens were written and read as JSON from a dedicated file. Cleanup was handled by writing an empty object into the file. Compared to key-value-based storage mechanisms, FileSystem has additional boilerplate and complexity, which increase the likelihood of implementation errors. Therefore, API complexity and configuration overhead are evaluated as medium.

FileSystem is only supported on native mobile platforms and it is not available in browser environments. To be able to provide equivalent token storage behavior on the web requires substituting FileSystem with an alternative storage mechanism, such as Web Storage APIs. This leads to a medium cross-platform implementation effort. As a result, the overall implementation cost of FileSystem is evaluated as medium.

Storage Mechanism	Confid.	API Complexity	Config. Overhead	Cross-platform effort	Overall Cost
In-memory	Medium	Low	Low	Low	Low
Web Storage APIs	Low	Low	Low	Medium	Low
IndexedDB	Low	Medium	Medium	Medium	Medium
Cookies (HttpOnly)	High	Low	High	High	High
AsyncStorage	Low	Low	Low	Low	Low
SecureStore	High	Low	Low	High	Medium
SQLite	Low	Medium	Medium	Medium	Medium
FileSystem	Low	Medium	Medium	Medium	Medium

Table 4.3: Comparison of token storage mechanisms by confidentiality and implementation cost

4.3 Discussion

The comparative evaluation of token storage mechanisms shows a clear trade-off between confidentiality and implementation cost. Only `HttpOnly` cookies in web environments and the SecureStore on mobile have high confidentiality, since they both protect tokens by default through isolation or encryption. In contrast, most

other mechanisms offer limited protection, since they persist tokens in locations accessible at runtime with no built-in protection. An exception is in-memory storage, which shows a medium level of confidentiality by greatly limiting the token exposure window.

From an implementation perspective, simpler key-value mechanisms such as in-memory storage and AsyncStorage offer low API complexity, low configuration overhead and low cross-platform implementation effort. In contrast, mechanisms that provide stronger confidentiality and security guarantees require additional infrastructure or platform-specific logic. For example, cookies require a backend component for secure session management and SecureStore does not offer equivalent functionality for web environments, meaning it would require a separate, more costly solution there.

More complex persistence mechanisms such as IndexedDB, SQLite and file-based storage do not provide meaningful benefits regarding confidentiality compared to simpler alternatives. At the same time, they introduce additional implementation complexity and overhead. In the context of this evaluation, these mechanisms do not offer a clear advantage for storing tokens in hybrid applications.

In answer to **RQ2**, no single token storage mechanism simultaneously provides both a high level of confidentiality and low implementation costs on both platforms. The most secure options, such as `HttpOnly` cookies and SecureStore require the most effort to implement, while simpler mechanisms, such as in-memory storage and AsyncStorage offer low costs but have the drawback of weaker confidentiality. In-memory storage may serve as a compromise, but it introduces further considerations regarding user experience, since tokens do not persist across sessions and remain accessible at runtime during the session.

The choice of storage mechanism should be guided by the application's threat model and platform requirements. For high security applications, a combination

of cookies on web environments and secure storage APIs on mobile platforms is recommended. In contrast, for scenarios where development speed and simplicity are prioritized, such as in prototyping an application, in-memory storage or a general purpose storage, such as AsyncStorage may be sufficient.

5 A Secure Token Management Architecture

This chapter addresses **RQ3**: "How can a secure and maintainable token management solution be implemented for hybrid applications?". The goal is to define a practical architecture and to demonstrate how it can be implemented. The solution is developed by using a design science approach, in which the architecture is built and analyzed based on the security and maintainability goals defined in Section 5.1.

5.1 Design Goals and Requirements

The evaluation done in Chapter 4 showed that no single token storage mechanism simultaneously provides both a high level of confidentiality and low implementation costs on both platforms. Storage mechanisms that offer high confidentiality, such as `HttpOnly` cookies in browser environments and secure storage APIs, such as SecureStore on mobile platforms, require platform-specific handling. In contrast, mechanisms that are simple to implement and reusable across platforms expose tokens to client-side code and provide limited protection. Since the trade-off between security and implementation cost is unavoidable, the main goal is to focus on a solution that prioritizes security. However, the implementation should be as maintainable as possible, in order to minimize implementation cost.

Maintainability in this thesis's context refers to an implementation that:

- Provides clear separation of concerns between application and authentication logic.
- Uses simple and consistent interfaces and/or abstractions
- Minimizes platform-specific code
- Is open for extension but closed for modification

The implementation must use the most secure storage mechanisms available on each platform. Tokens are stored using `HttpOnly` cookies in web environments and natively supported secure storage solutions in mobile environments. In addition to these considerations, the design must follow best practices for OAuth 2.0 and related specifications.

5.2 System Architecture

The architecture consists of three primary elements: a hybrid application, a backend service and an identity provider (IdP). A high level overview of the architecture is illustrated in Figure 5.1. The hybrid application uses a single codebase, which includes implementations for both native mobile and browser-based platforms. The hybrid application is responsible for user interaction and initiating authorization flows. The mobile implementation acts as a public client and communicates directly with the IdP by using the Authorization Code Flow with PKCE. Tokens that are issued to the mobile implementation are stored using native secure store mechanisms. In contrast, the browser-based implementation delegates authentication and token management to the backend service, which also enforces authorization, provides resources and works as a confidential client. The backend performs the Authorization Code Flow with PKCE enabled, maintains the authenticated session on behalf of the web application and issues `HttpOnly` cookies back to the browser.

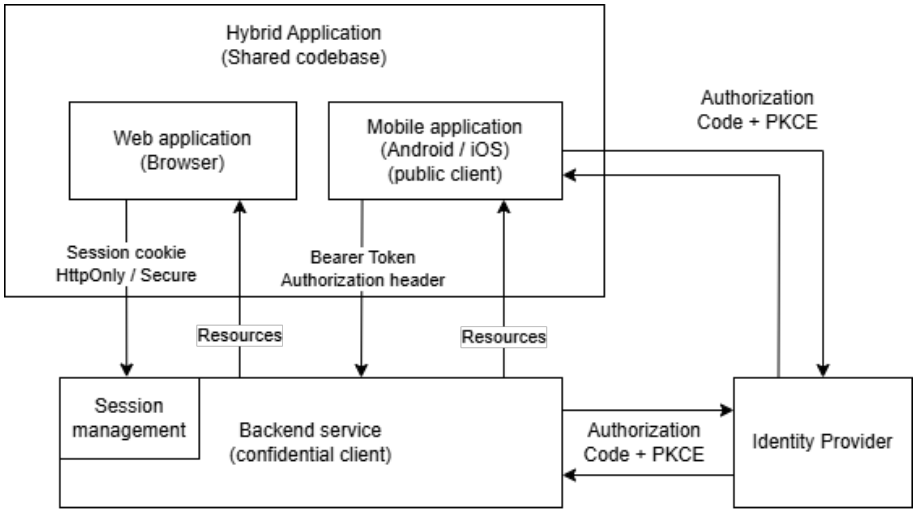

Figure 5.1: High level architecture

5.3 Implementation

5.3.1 Technologies

The system's parts are implemented by using Expo [2] as the hybrid application framework, a Quarkus [54] backend service and Keycloak [53] as the IdP. Expo enables for a shared React Native codebase for both mobile and web environments. The backend service is implemented using Quarkus with Java, which provides REST APIs to expose protected resources, session handling where applicable and integration with the IdP. Identity and access management is provided by Keycloak, which is responsible for authentication, authorization and issuing tokens. The implementation resides on GitHub: `https://github.com/Uteeaami/secure-hybrid-auth-storage`.

5.3.2 Hybrid Application

The hybrid application's authentication related functionality is encapsulated within an `AuthProvider` component. This provider exposes a minimal interface that consists of login, logout, loading state and authentication state. The provider isolates

authentication logic from application views, which allows UI components to remain separate from the authentication mechanism. Since both token storage and authentication approaches differ between platforms, the provider is designed to handle both. The application dynamically selects the appropriate token storage mechanism and authentication behavior based on the active environment. Upon application startup, the provider performs an automatic session validation via the backend `/session` endpoint. During logout, the application calls either the IdP's token revocation endpoint or the backend service's `/logout` endpoint.

Mobile Environment

In the mobile environment, the application operates as a public client. It communicates directly with the IdP by using the Authorization Code Flow with PKCE. The authentication process is initiated when a user taps the 'Login' button. This initiates an authorization request which includes the PKCE `code_challenge` and opens the system browser. After successful authentication, the IdP returns an authorization code to the application. The application then exchanges the authorization code for tokens using the PKCE `code_verifier`. Tokens are stored, by using Expo SecureStore. The authentication process in the mobile environment is illustrated in Figure 5.2.

Web Environment

In the web environment, token management and authentication are delegated to the backend service, which acts as a confidential client. The backend communicates with the IdP using the Authorization Code Flow with PKCE enabled. Instead of exposing tokens to the browser, they are stored server-side. When the web application initiates login, the backend starts an authorization request which includes the PKCE `code_challenge` and redirects the browser to the IdP's login page. After

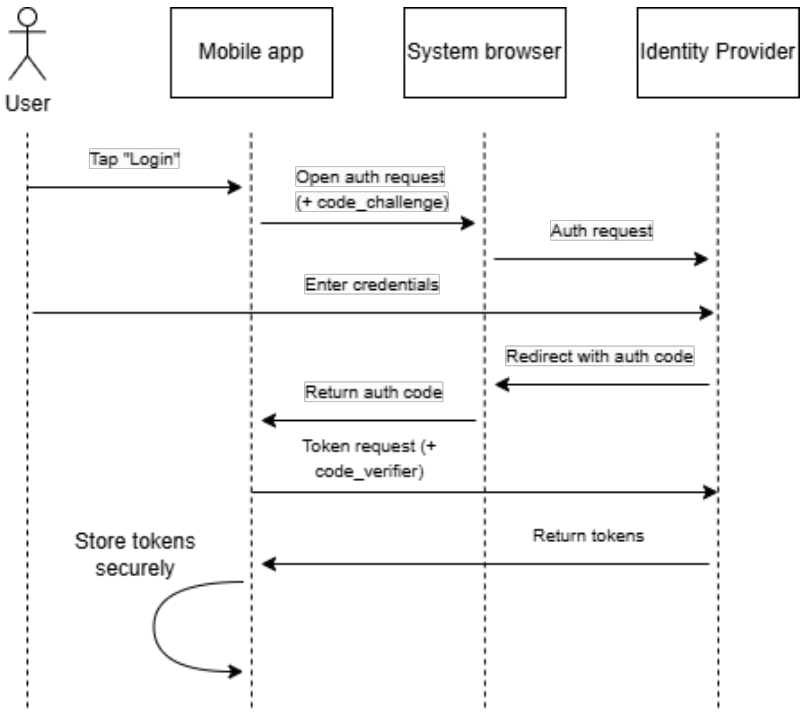

Figure 5.2: Authentication process in mobile environment

successful authentication, the IdP returns an authorization code to the backend service, which exchanges it for tokens with the PKCE `code_verifier`. The backend service then creates an authenticated session and issues a session cookie configured with `HttpOnly`, `Secure` and `SameSite` attributes. The authentication process in the web environment is illustrated in Figure 5.3.

API Communication

Since the hybrid application must be capable of retrieving resources from the backend service, the API requests are handled differently in each environment. Requests are made through an `authFetch` wrapper utility, which isolates platform-specific authentication behavior while providing a consistent interface. In the mobile environment, `authFetch` attaches the access token as a Bearer token in the Authorization header of outgoing requests. Furthermore, in the mobile environment `authFetch` is responsible for triggering the token refresh logic, given that an unauthorized re-

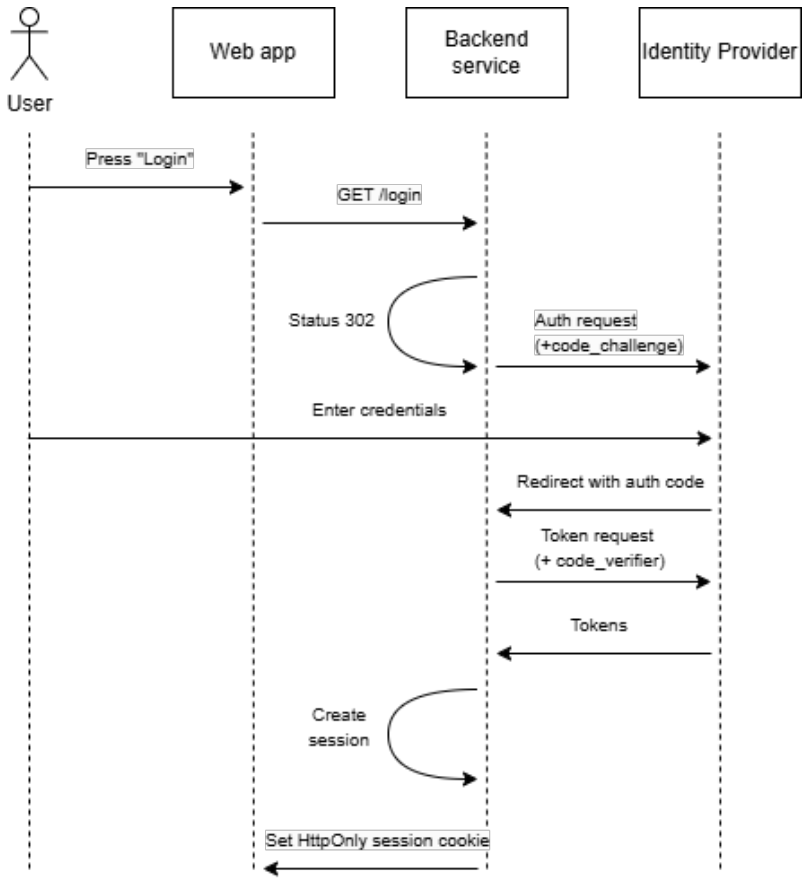

Figure 5.3: Authentication process in web environment

sponse is returned. In the web environment, authentication relies on automatically included session cookies.

5.3.3 Backend Service

The backend service is implemented by using Quarkus. Its primary responsibility is to eliminate the browser-based client's need to interact with tokens. Instead, tokens and the authenticated session are managed by the backend service, which associates them with secure session cookies issued to the client. The backend service operates as a confidential client and integrates with the IdP by using the Quarkus OIDC extension. Authentication is performed by using the Authorization Code Flow with PKCE enabled.

OIDC Integration and Configuration

The backend service is configured as an OIDC `hybrid` application type. In Quarkus terminology, this configuration allows the application to function simultaneously as a web application and as a protected resource server. This means that when an incoming request contains a Bearer token in the Authorization header, the service validates the token and processes the request as an API call. Otherwise, unauthenticated browser requests trigger the Authorization Code Flow. [61] The configuration defines the IdP endpoint, client credentials, cookie attributes and authentication policies. The most important configuration details include:

- Session cookies configured with `HttpOnly`, `Secure`, and `SameSite` attributes
- PKCE for authorization requests
- Issuer validation
- Audience restriction to specific clients
- CORS restrictions that limit requests to the desired origin

Authentication and Session Management

Authentication starts through the `/api/v1/auth/login` endpoint. When this endpoint is accessed by an unauthorized user, the backend service redirects the browser to the IdP's login page. After successful authentication, the IdP returns an authorization code to the backend service, which then exchanges it for tokens and starts an authenticated session. Tokens stored by the Quarkus OIDC extension are encrypted by default when the standard configuration is used. The session lifecycle is managed through the `OidcSession` interface, which is provided by the extension. The extension also refreshes tokens in the background using the configured time skew until

the access token expires. Sessions are tied to browser cookies and included in subsequent requests. Furthermore, a dedicated public session endpoint `/api/v1/session` is exposed to allow the hybrid application to verify whether an authenticated session exists. Logout functionality is implemented by the `/api/v1/auth/logout` endpoint. The backend service terminates the session by clearing session cookies and revokes tokens through the IdP's revocation endpoint.

5.3.4 Identity Provider

The identity provider is implemented using Keycloak, which is responsible for authenticating users, issuing tokens and enforcing authorization policies. The IdP defines two separate OAuth client configurations under the same realm. Both clients have the Authorization Code Flow enabled and other flows are disabled. Each client has its own redirect URIs and uses S256 as the PKCE code challenge method. The first client is configured as a public client, which represents the mobile application. The second is a confidential client, which represents the backend service. Token lifetimes, scopes and revocation are controlled by the IdP under the same realm, which allows for consistent enforcement on both platforms. The access token lifetime in this implementation is five minutes, and the refresh token lifetime is ten minutes. These lifetimes are intentionally short. In a production environment, refresh token lifetimes would typically be considerably longer, such as hours or days. Furthermore, refresh tokens are rotated and can only be used once.

5.4 Security Analysis

Security related risks mainly come from environments that are more public-facing than others. In these environments, application code can be inspected or manipulated by an attacker, such as browser-based clients in web environments. In the

context of this thesis, the different components within the system can be separated into three levels of trust:

- Web environment (untrusted)
- Mobile environment (partially trusted)
- Backend service and identity provider (trusted)

The web environment is considered untrusted because the application executes inside the user's browser, which is outside the system's control. These applications can be inspected using developer tools and client-side code can potentially be manipulated by an attacker. Additionally, browsers are exposed to common web-based vulnerabilities such as XSS and CSRF.

In contrast, the mobile environment is considered partially trusted because operating system level protections, such as application-level sandboxing, hardware-backed secure storage and biometric authentication provide stronger isolation than the browser environment. However, the device itself cannot be fully trusted because it may be compromised, rooted or manipulated by the user or by other malicious applications.

The backend service and IdP are considered trusted components since they commonly operate in a controlled server-side environment. In these environments the application code, infrastructure and security configurations are adjustable and under administrative control.

One of the primary concerns in hybrid applications is the exposure of tokens. In the web environment, the implemented architecture prevents tokens from being stored in the browser. Tokens remain exclusively in the backend service in encrypted form and the authentication state is forwarded to the browser via `HttpOnly` cookies. This eliminates the risk of XSS and insecure storage threats, which were identified as the highest risk threats in Section 3.2. Additionally, there are token validation

mechanisms which are enforced by the backend service to ensure that only legitimate tokens are accepted. This is done by verifying the issuer `iss` and audience `aud` claims. In the mobile environment, tokens are stored using operating system backed secure storage mechanisms. These environments provide hardware-backed protection and an application level sandbox to address the threat of insecure storage.

Both mobile and web authentication processes use the Authorization Code Flow with PKCE, which mitigates the threat of authorization code interception. Even though the backend service operates as a trusted, confidential client PKCE is still used, which enhances security and aligns with the known best practices mentioned by specifications such as OAuth 2.1 [17]. Exact matching of redirect URIs is also used to further reduce the risk of authorization code interception and redirect URI manipulation.

The authenticated sessions are bound to secure cookies and managed by the backend service in the web environment. In the mobile environment the session is handled client-side. Tokens are discarded and revoked upon logout, which ensures that the authentication state is invalidated. Additionally, tokens are short-lived and refresh tokens are rotated and can be used only once. These practices address the insufficient token lifecycle management threat, which was identified in Section 3.2.

In addition to these mechanisms, the web environment addresses the CSRF threat. The `SameSite` attribute set to `Strict` or `Lax` mitigates CSRF by restricting cross-site requests and CSRF tokens provide additional protection. MITM attacks should be addressed in both environments through TLS and its use is enforced by the cookies' `Secure` attribute in the web environment. Furthermore, in the mobile environment, instead of using embedded views for authentication, the application uses the system browser, which reduces the risk of credentials being leaked.

5.5 Maintainability Analysis

One of the main high level maintainability benefits is the use of the hybrid application approach. Both the web and mobile implementations share a codebase for a significant amount of the application logic and user interface components. This reduces the need for code duplication and allows new features to be implemented once and be used across all platforms. However, hybrid applications introduce platform-specific challenges that may require their own solutions, such as the underlying authentication logic. These solutions should be managed carefully and be isolated as much as possible from other application logic. They should also be accessed through well-defined interfaces in order to avoid unnecessary complexity in the shared codebase. For example, in this implementation the authentication logic is handled through a single `AuthProvider` component that determines how the authentication process should be executed depending on the environment. No other conditional logic or components should be able to intercept or modify this process. Furthermore, the `authFetch` wrapper has a similar role in the request context. The wrapper ensures that authenticated requests are handled consistently across environments.

The backend service acts as a central component in the system architecture. The backend service handles authentication logic, session management and authorization decisions, which keeps the web and mobile clients relatively simple in this regard. Furthermore, the backend service relies on well-established framework extensions to handle encryption, integration with the IdP and PKCE. Therefore, many concerns related to security are configuration details rather than custom application logic, which can improve maintainability. Additionally, the backend service works as an intermediary for the different clients. If the system were to be expanded with additional services, which is very common in microservices architectures, the backend service could behave as an intermediary between the clients and the new services that provide other resources. This can further improve both maintainability and

scalability.

Despite the many benefits of this approach, this architecture introduces complexities. The use of multiple components increases the overall system complexity compared to simpler monolithic applications. A simpler solution could be to store tokens directly in client-side storage in the web environment, which would eliminate the need for a backend intermediary. However, this approach would introduce additional security risks. Therefore, the added architectural complexity is justified by the improved security.

Overall, the presented architecture and its implementation provides good maintainability characteristics. Platform-specific code is minimized and there is clear separation of concerns between application logic and authentication logic, which allows for future extension at both the code and system levels.

5.6 Discussion

The presented architecture improves security, but it also increases complexity. A separate backend service means that additional infrastructure must be deployed, maintained and monitored. In contrast, a simpler architecture that would only consist of a hybrid application and an identity provider would require fewer components and would be less complex. However, this simpler approach would require tokens to be stored in a less secure manner, especially in the web environment. The additional components increase the overall system complexity, but they allow for handling sensitive data and other security related operations within a trusted environment. This significantly improves the overall security of the system and still benefits from the hybrid application development model.

Another important aspect is that the web and mobile clients should be treated as separate entities, even though they share largely the same codebase. From an authentication perspective they behave as distinct clients. However, security policies

and configuration settings such as access token lifetime should be defined consistently for both clients. In practice, these configurations are typically defined at the realm or organization level within the IdP and inherited by individual client configurations.

The architecture supports future extension. Additional backend services can be introduced without requiring major changes to the client implementations. The system could evolve toward a more distributed architecture if needed, where the separate backend service would act as an intermediary between the hybrid application and other services.

An alternative approach for implementing the backend component could also be considered and explored further. For example, the Expo framework provides API routes that could be used to implement authentication logic directly in the hybrid application's codebase. API routes are a way to write secure server-side code [62]. Conceptually, this would function similarly to the backend service implemented in this work. API routes would act as a secure server-side component and it would be responsible for handling authentication sessions, storing tokens and providing `HttpOnly` session cookies for the web environment. This approach would simplify the provided implementation by removing the need for a separate backend codebase. However, this was not explored further in this work because it would require implementing several security related features manually, such as PKCE code generation, token refresh logic and other security related functionalities like encryption. In contrast, the Quarkus OIDC extension provides these features out of the box. Nevertheless, the chosen technology is only an implementation detail. The overall architecture, token storage mechanisms and client-side authentication logic would remain largely the same regardless of the technologies used.

It is also worth noting that storing the ID token on the client-side may not always be necessary. User profile information can be retrieved from the `userinfo` endpoint when needed, which avoids unnecessary storage of ID tokens. This reduces the

amount of sensitive user information stored on the client, which reduces potential privacy related risks in case of token leakage.

In answer to **RQ3**, a secure and maintainable solution can be achieved by combining a backend-managed session with `HttpOnly` cookies for the web environment and operating system backed secure storage for the mobile environment. Overall, the proposed architecture represents a trade-off between implementation cost and security. Security could be further improved through mechanisms such as proof-of-possession (DPoP), which associate issued tokens to a specific client. This would prevent stolen tokens from being reused in a different environment. These mechanisms were not included in this architecture since the primary focus of this thesis is on the token storage mechanism.

6 Conclusion

This thesis examined the security of token management in hybrid applications. The goal was to identify the most significant threats that relate to tokens, evaluate different token storage mechanisms and propose a secure and maintainable token management solution for hybrid applications.

Regarding the first research question, the threat analysis showed that tokens are exposed to multiple relevant threats in hybrid environments. The most significant risks come from threats that can directly expose tokens, such as XSS and insecure storage. These threats were evaluated as high risk, since they have a high likelihood and impact. Other threats related to direct exposure, such as phishing and leakage were evaluated as medium to high risk, while device theft was evaluated as medium risk. Additionally, protocol and configuration related threats, such as authorization code interception were evaluated as medium to high risk, while insufficient token lifecycle management and CSRF were evaluated as medium risk. Finally, threats that target the underlying environment or infrastructure, such as MITM attacks, malware infections and software supply chain attacks, were evaluated as medium to high risk. Overall, the evaluation shows that the primary security concern for tokens in hybrid applications is their possible exposure rather than availability.

The second research question evaluated how different token storage mechanisms compare in terms of confidentiality and implementation cost in hybrid applications. There is a clear trade-off between these two aspects. Storage mechanisms that pro-

vide stronger confidentiality, such as `HttpOnly` cookies in web environments and the secure storage APIs, such as Expo's SecureStore on mobile platforms, require additional infrastructure or platform-specific implementation to achieve equal level of functionality on the other platform. In contrast, simpler mechanisms such as in-memory or AsyncStorage have low implementation cost, but they provide weaker protection because tokens remain accessible at runtime. In general the evaluation shows that no single storage mechanism simultaneously provides strong confidentiality guarantees and low implementation cost on both platforms.

Finally, the third research question focused on how a secure and maintainable token management architecture can be implemented for hybrid applications. The solution separates responsibilities between the hybrid application and a backend service. In this architecture, the backend service acts as a confidential OAuth client for the web environment and manages authenticated session using `HttpOnly` cookies. In contrast, the mobile client stores tokens using platform-specific secure storage. Although this solution introduces additional complexity, it allows token management to be performed within a more trusted environment.

This work has some limitations. The evaluation of token storage mechanisms was qualitative and focused mainly on confidentiality and implementation cost instead of a empirical evaluation. Future work could do empirical studies related to the different token storage mechanisms, especially on the mobile environment side.

In conclusion, protecting tokens in hybrid applications requires careful consideration of security threats and limitations set up by the development context. Not a single storage mechanism provides an ideal solution across all environments. Practical balance between overall security, confidentiality and maintainability can be achieved by combining backend session handling for web environments and secure storage APIs on mobile environments.

References

[1] *Cordova*, https://cordova.apache.org/, Accessed: 2026-03-22.

[2] *Expo*, https://expo.dev/, Accessed: 2026-03-01.

[3] *Ionic*, https://ionicframework.com/, Accessed: 2026-03-22.

[4] *Flutter*, https://flutter.dev/, Accessed: 2026-03-22.

[5] K. Kishore, S. Khare, V. Uniyal, and S. Verma, "Performance and stability comparison of react and flutter: Cross-platform application development", in *2022 International Conference on Cyber Resilience (ICCR)*, 2022, pp. 1–4. DOI: 10.1109/ICCR56254.2022.9996039.

[6] P. Mishra, S. Sachdeva, A. Kumar, and A. Kumar, "Hybrid application development and implementation", in *2019 6th International Conference on Computing for Sustainable Global Development (INDIACom)*, 2019, pp. 102–107.

[7] *About the new architecture*, Accessed: 2025-10-14. [Online]. Available: https://reactnative.dev/architecture/landing-page.

[8] *Web view*, Accessed: 2025-10-21. [Online]. Available: https://ionicframework.com/docs/core-concepts/webview.

[9] *Flutter architectural overview*, Accessed: 2025-11-07. [Online]. Available: https://docs.flutter.dev/resources/architectural-overview.

[10] A. Jøsang, "A consistent definition of authorization", in *Security and Trust Management*, G. Livraga and C. Mitchell, Eds., Cham: Springer International Publishing, 2017, pp. 134–144.

[11] *What is access control?*, Accessed: 2025-10-21. [Online]. Available: https://www.microsoft.com/en-us/security/business/security-101/what-is-access-control.

[12] *Authentication vs. authorization*, Accessed: 2025-10-22. [Online]. Available: https://learn.microsoft.com/en-us/entra/identity-platform/authentication-vs-authorization.

[13] D. Hardt, "The oauth 2.0 authorization framework", RFC Editor, Tech. Rep. 6749, 2012, Accessed: 2025-09-08. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6749.txt.

[14] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore, "Openid connect core 1.0 incorporating errata set 2", OpenID Foundation, Tech. Rep., 2023, Accessed: 2025-09-08. [Online]. Available: https://openid.net/specs/openid-connect-core-1_0.html.

[15] *Authentication with OAuth or OpenID providers*, Accessed: 2026-03-22. [Online]. Available: https://docs.expo.dev/guides/authentication/.

[16] *Auth Connect*, Accessed: 2026-03-22. [Online]. Available: https://ionic.io/docs/auth-connect.

[17] D. Hardt, A. Parecki, and T. Lodderstedt, "The oauth 2.1 authorization framework", IETF Secretariat, Tech. Rep. draft-ietf-oauth-v2-1-13, 2025, Accessed: 2025-09-08. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-oauth-v2-1/.

[18] M. B. Jones and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC Editor, Tech. Rep. 6750, 2012, Accessed: 2025-04-11. [Online]. Available: https://www.rfc-editor.org/info/rfc6750.

[19] A. Parecki, *Scope*, Accessed: 2025-10-26. [Online]. Available: https://www.oauth.com/oauth2-servers/scope/.

[20] N. Sakimura, J. Bradley, and N. Agarwal, "Proof key for code exchange by oauth public clients", RFC Editor, Tech. Rep. 7636, 2015, Accessed: 2025-09-08. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7636.

[21] M. Nieles, K. Dempsey, and V. Pillitteri, "An introduction to information security", National Institute of Standards and Technology, Tech. Rep., 2017, Accessed: 2025-11-04, pp. 1–6. [Online]. Available: https://csrc.nist.gov/pubs/sp/800/12/r1/final.

[22] J. Cawthra, M. Ekstrom, L. Lusty, J. Sexton, J. Sweetnam, and A. Townsend, "Nist special publication 1800-26a", National Institute of Standards and Technology, Tech. Rep., 2020, Accessed: 2025-11-04. [Online]. Available: https://www.nccoe.nist.gov/publication/1800-26/VolA/index.html.

[23] A. van der Stock B. Glas N. Smithline T. Janca T. Gigler, *OWASP Top 10*, Accessed: 2025-11-10, 2025. [Online]. Available: https://owasp.org/Top10/2025/0x00_2025-Introduction/.

[24] *OWASP Mobile Top 10*, Accessed: 2025-11-10, 2024. [Online]. Available: https://owasp.org/www-project-mobile-top-10/.

[25] *Web Storage API*, Accessed: 2025-11-17. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API.

[26] *Using http cookies*, Accessed: 2025-11-18. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Cookies.

[27] A. Barth, "HTTP State Management Mechanism", RFC Editor, Tech. Rep. 6265, 2011, Accessed: 2025-11-18. [Online]. Available: https://www.rfc-editor.org/info/rfc6265.

[28] *Indexeddb api*, Accessed: 2025-11-18. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API.

[29] *Expo SecureStore*, Accessed: 2025-09-16. [Online]. Available: https://docs.expo.dev/versions/latest/sdk/securestore/.

[30] *Avainnipun tietojen suojaus*, Accessed: 2025-09-16. [Online]. Available: https://support.apple.com/fi-fi/guide/security/secb0694df1a/web.

[31] *Android Keystore system*, Accessed: 2025-09-16. [Online]. Available: https://developer.android.com/privacy-and-security/keystore.

[32] *Expo SQLite*, Accessed: 2025-11-20. [Online]. Available: https://docs.expo.dev/versions/latest/sdk/sqlite/.

[33] *Expo FileSystem*, Accessed: 2025-11-20. [Online]. Available: https://docs.expo.dev/versions/latest/sdk/filesystem/.

[34] *Async Storage*, Accessed: 2025-09-16. [Online]. Available: https://react-native-async-storage.github.io/async-storage/.

[35] *React Native Async Storage*, Accessed: 2025-11-20. [Online]. Available: https://react-native-async-storage.github.io/3.0-next/.

[36] *React native directory*, Accessed: 2025-11-20. [Online]. Available: https://reactnative.directory/?search=storage.

[37] V. Drake, *Threat Modeling*, Accessed: 2025-11-20. [Online]. Available: https://owasp.org/www-community/Threat_Modeling.

[38] *Microsoft Threat Modeling Tool threats*, Accessed: 2025-12-09. [Online]. Available: https://learn.microsoft.com/en-us/azure/security/develop/threat-modeling-tool-threats.

[39] T. Lodderstedt, M. McGloin, and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations", RFC Editor, Tech. Rep. 6819, 2013, Accessed: 2025-12-09. [Online]. Available: https://www.rfc-editor.org/info/rfc6819.

[40] T. Lodderstedt, J. Bradley, A. Labunets, and D. Fett, "Best Current Practice for OAuth 2.0 Security", RFC Editor, Tech. Rep. 9700, 2025, Accessed: 2025-11-09. [Online]. Available: https://www.rfc-editor.org/info/rfc9700.

[41] A. Parecki, P. D. Ryck, and D. Waite, "OAuth 2.0 for Browser-Based Applications", Internet Engineering Task Force, Tech. Rep., 2025, Accessed: 2025-09-10. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-oauth-browser-based-apps/25/.

[42] W. Denniss and J. Bradley, "OAuth 2.0 for Native Apps", RFC Editor, Tech. Rep. 8252, 2017, Accessed: 2025-12-09. [Online]. Available: https://www.rfc-editor.org/info/rfc8252.

[43] *Cross-site scripting (xss)*, Accessed: 2025-09-10. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Security/Attacks/XSS.

[44] K. Haider, F. Nasim, and S. Ali, "Mitigating local storage and session storage vulnerabilities through secure middleware", *Al-Aasar*, vol. 2, no. 1, pp. 523–540, Apr. 2025. [Online]. Available: https://al-aasar.com/index.php/Journal/article/view/213.

[45] A. van der Stock B. Glas N. Smithline T. Janca T. Gigler, *A05:2025 Injection*, Accessed: 2025-03-22, 2025. [Online]. Available: https://owasp.org/Top10/2025/A05_2025-Injection/.

[46] *M2: Insecure Data Storage*, Accessed: 2025-12-01, 2024. [Online]. Available: https://owasp.org/www-project-mobile-top-10/2016-risks/m2-insecure-data-storage.

[47] *M1: Improper Credential Usage*, Accessed: 2026-04-15, 2024. [Online]. Available: https://owasp.org/www-project-mobile-top-10/2023-risks/m1-improper-credential-usage.html.

[48] *Transport Layer Security Cheat Sheet*, Accessed: 2025-12-14. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Transport_Layer_Security_Cheat_Sheet.html.

[49] A. van der Stock B. Glas N. Smithline T. Janca T. Gigler, *A03:2025 Software Supply Chain Failures*, Accessed: 2025-03-10, 2025. [Online]. Available: https://owasp.org/Top10/2025/A03_2025-Software_Supply_Chain_Failures/.

[50] *M6: Inadequate Privacy Controls*, Accessed: 2026-04-15, 2024. [Online]. Available: https://owasp.org/www-project-mobile-top-10/2023-risks/m6-inadequate-privacy-controls.html.

[51] I. Darmawan, A. P. A. Karim, A. Rahmatulloh, R. Gunawan, and D. Pramesti, "Json web token penetration testing on cookie storage with csrf techniques", in *2021 International Conference Advancement in Data Science, E-learning and Information Systems (ICADEIS)*, 2021, pp. 1–5. DOI: 10.1109/ICADEIS52521.2021.9701965.

[52] *OAuth 2.0 Protocol Cheatsheet*, Accessed: 2025-11-09. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/OAuth2_Cheat_Sheet.html.

[53] *Keycloak*, https://www.keycloak.org/, Accessed: 2026-03-01.

[54] *Quarkus*, https://quarkus.io/, Accessed: 2026-03-01.

[55] *Activity lifecycle*, https://developer.android.com/guide/components/activities/activity-lifecycle, Accessed: 2026-01-24.

[56] *Managing your app's life cycle*, https://developer.apple.com/documentation/uikit/managing-your-app-s-life-cycle, Accessed: 2026-01-24.

[57] *Mobile application security cheat sheet*, https://cheatsheetseries.owasp.org/cheatsheets/Mobile_Application_Security_Cheat_Sheet.html, Accessed: 2026-01-24.

[58] *Application sandbox*, https://source.android.com/docs/security/app-sandbox, Accessed: 2026-03-19.

[59] *App sandbox*, https://developer.apple.com/documentation/security/app-sandbox, Accessed: 2026-03-19.

[60] *Cross-site request forgery prevention cheat sheet*, https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html, Accessed: 2026-01-25.

[61] *Quarkus openid connect (oidc) expanded configuration reference*, https://quarkus.io/guides/security-oidc-expanded-configuration, Accessed: 2026-03-05.

[62] *Api routes*, https://docs.expo.dev/router/web/api-routes/, Accessed: 2026-03-08.

Appendix A Use of Generative AI

Generative AI tools were used to improve text quality and fluency by reviewing the already written text without changing the original content. The suggested changes were compared with the original text and applied if they improved the overall text quality and fluency.