Dipartimento di Informatica, Bioingegneria,
Robotica ed Ingegneria dei Sistemi

Analysis of Best Current Practices to Assist Native App Developers with Secure OAuth/OIDC Implementations

by

Amir Sharif

Theses Series **DIBRIS-TH-2021-07**

DIBRIS, Università di Genova
Via Opera Pia, 13 16145 Genova, Italy http://www.dibris.unige.it/

Università degli Studi di Genova

Dipartimento di Informatica, Bioingegneria,

Robotica ed Ingegneria dei Sistemi

Ph.D. Thesis in Computer Science and Systems Engineering
Computer Science Curriculum

Analysis of Best Current Practices to Assist Native App Developers with Secure OAuth/OIDC Implementations

by

Amir Sharif

July, 2021

Dottorato di Ricerca in Informatica ed Ingegneria dei Sistemi
Indirizzo Informatica
Dipartimento di Informatica, Bioingegneria, Robotica ed Ingegneria dei Sistemi
Università degli Studi di Genova

DIBRIS, Univ. di Genova
Via Opera Pia, 13
I-16145 Genova, Italy
http://www.dibris.unige.it/

Ph.D. Thesis in Computer Science and Systems Engineering
Secure and Reliable Systems Curriculum
(S.S.D. ING-INF/05, INF/01)

Submitted by Amir Sharif
DIBRIS, Univ. di Genova
Security & Trust Research Unit, FBK-Irst, Trento
asharif@fbk.eu, S4543475@studenti.unige.it

Date of submission: May 2021

Title: Analysis of Best Current Practices to Assist Native App Developers with Secure OAuth/OIDC Implementations

Advisor: Roberto Carbone[1]
Co-Advisors: Silvio Ranise,[1,2] Giada Sciarretta[1]
[1]Security & Trust Research Unit, FBK-Irst, Trento (Italy)
[2]Department of Mathematics, University of Trento, Trento (Italy)

Ext. Reviewers: Stefano Calzavara,[†] Gabriele Costa[*]
[†]Department of Environmental Science, Informatics and Statistics, Ca' Foscari University, Venice (Italy)
[*]IMT School for Advanced Studies, Lucca (Italy)

Abstract

OAuth 2.0 and OpenID Connect are two of the most widely used protocols to support secure and frictionless access delegation and single sign-on login solutions, which have been extensively integrated within web and mobile native applications. While securing the OAuth and OpenID Connect implementations within the web applications is widely investigated, this is not true for mobile native applications due to their peculiarities compared to web applications. Given that, we investigate the availability of necessary information to mobile native application developers. Our investigation reveals that mobile native application developers need to access many sparse documents and understand technical security writing, when they are not necessarily security experts that leads to insecure integration of OAuth and OpenID Connect solutions due to various implementation flaws. Thus, to assist mobile native application developers in the understanding of OAuth and OpenID Connect documentations, we demystify the OAuth and OpenID Connect core documentations and two of the most security critical profiles for governmental and financial domains, namely "International Government Assurance" and "Financial Grade API" to extract the wealth information and summarize them in plain English.

To secure the integration of OAuth and OpenID Connect solutions, the OAuth working group and the OpenID foundation have produced many security-related documents to provide general guidelines and best current practices. These documents explain the features that OAuth and OpenID Connect providers must support and how web and mobile native application developers should implement these solutions for the different use case scenarios. In addition, due to the peculiarities of mobile native applications, the OAuth working group has published the "OAuth 2.0 for Native Apps" documentation dedicated to assist mobile native application developers. Recently, the OAuth working group released AppAuth SDK to support mobile native application developers in the secure implementation of access delegation and single sign-on login solutions within mobile native applications. It enables mobile native applications to authorize and authenticate users by communicating with OAuth and OpenID Connect providers, beside embedding the security and usability best current practices described in [DB17]. We thus perform a comprehensive analysis to inves-

tigate the compliance with the best current practices of the main OAuth and OpenID Connect providers and top-ranked Google Play Store applications. Our analysis shows that 7 out of 14 providers, and 5 out of 87 top-ranked Google Play Store applications are fully compliant with the best current practices and none of the Google Play Store applications use AppAuth SDK.

We conjecture that the root-causes of the non-compliant solutions are different for OAuth and OpenID Connect providers and Google Play Store applications. Concerning providers, they might be aware of these best current practices violations and their non-compliant solutions can be due to legacy reasons. Concerning Google Play Store applications, their non-compliant solutions can be due to the following: (i) the best current practices documents for OAuth and OpenID Connect are sparse, and mobile native application developers may be either unaware of them or misinterpret them as they are not (necessarily) security expert, (ii) lack of the best current practices adoption by OAuth and OpenID Connect providers that leads to the difficulty in integration of AppAuth SDK within mobile native applications. In addition, even in the case of compliant OAuth and OpenID Connect providers, the mobile native application developers still need to properly configure the AppAuth SDK and write the secure code to invoke the SDK properly within their mobile native applications, which is a daunting task, and (iii) the pressure on mobile native application developers to provide new functionalities for the mobile native applications may result in prioritizing the functionality over the security—as performing a risk assessment procedure is a complex task in the context of OAuth and OpenID Connect solutions—they could not have the resources to perform a risk assessment procedure.

The above-mentioned problems motivate us to propose methodologies to assist mobile native application developers with the secure implementation of OAuth and OpenID Connect solutions within their mobile native applications. To this aim, we provide a reference model for OAuth and OpenID Connect solutions by utilizing the extracted information from various documents that can be used within a risk assessment approach to enable mobile native application developers with an informed decision w.r.t. their implementation choices. In addition, we design a wizard-based approach and implement it within an Android Studio plugin called mIDAssistant that assists mobile native application developers with automatic integration of the core functionalities and ensure the enforcement of the best current practices by leveraging AppAuth SDK.

The effectiveness of our approach has been verified in several real-world scenarios (e.g., pull printing), research and innovation projects (e.g., the EIT Digital activity API Assistant), and in the context of industrial collaborations (Poste Italiane, IPZS). Furthermore, we had the opportunity to present our work to the OAuth working group experts (during the OAuth Security Workshop), and they have shown interest

in our approach.

Acknowledgements

Firstly, I would like to express my sincere gratitude to my advisor Roberto Carbone, who has believed in me since the beginning of my research activities. I am very grateful for your support, patience, care, and discussions that help me do my best in every aspect of my research.

I would also like to pay my special regards to my co-advisors, Silvio Ranise and Giada Sciarretta for your unconditional supports during these years. Your comments and observations have always guided me towards the best way in my research.

I would also like to thank Prof. Giorgio Delzanno, coordinator of my Ph.D. Program, the technical committees and the reviewers.

I would like to thank my colleagues (Alex, Andrea, Biniam, Giada, Majid, Marco, Salimeh, Salvatore, Stefano and Umberto) in the Security & Trust Unit, where I have spent most of my time during my research activity. Special thanks go to Salimeh for the joyful collaboration when we were working together on an interesting interdisciplinary research topic.

Finally, I would like to thank my family, friends, and people who supported me during these years. I could not have made it without you.

Table of Contents

List of Figures **6**

List of Tables **8**

Chapter 1 Introduction **10**

1.1 Context and Motivations 10

1.2 Contributions 15

1.3 Structure of the Thesis 16

Chapter 2 Background **20**

2.1 Identity Management 20

2.2 IdM Standards 21

2.2.1 OAuth 2.0 21

2.2.2 OpenID Connect 23

2.2.3 Native Application Scenario 24

2.3 OAuth Extensions 25

2.3.1 Proof Key for Code Exchange (PKCE) 26

2.3.2 Dynamic Client Registration (DCR) 27

2.4 OAuth and OpenID Connect Profiles 28

2.4.1 International Government Assurance Profile (iGov) 28

2.4.2 Financial Grade API Profile (FAPI) 28

2.4.3 OpenID Connect for Identity Assurance Profile 29

Chapter 3 Demystify Best Current Practices for Native Apps 31

3.1 Context and Motivations 31

3.2 Threat Model 32

3.3 Best Current Practices Definition 34

3.3.1 OAuth/OIDC Security Best Current Practices 35

3.3.2 iGov Security Best Current Practices 40

3.3.3 FAPI Security Best Current Practices 44

3.3.4 Our Recommended Privacy Best Current Practices 47

Chapter 4 Compliance Analysis of IdMPs and Google Play Store Apps with OAuth and OIDC Security BCPs 54

4.1 Context and Motivations 54

4.2 Analysis of Popular OAuth/OIDC IdMPs 55

4.2.1 IdMPs Selection 55

4.2.2 Feature Definition 55

4.2.3 Research Questions and Results 56

4.3 Analysis of Top-Ranked Google Play Store Apps 61

4.3.1 Selection Criteria 61

4.3.2 Selected Features 64

4.3.3 Selected Static Analysis Tools 65

4.3.4 Research Questions and Results 66

Chapter 5 OAuth/OIDC Risk Assessment 72

5.1 Context and Motivations 72

5.2 OAuth/OIDC Reference Model 73

5.3 Problem definition 79

Chapter 6 mIDAssistant Plugin **83**

6.1 Wizard-based Approach 84

6.2 Support fully compliant IdMP 84

6.2.1 Security Experts: DB population 85

6.2.2 Developers: Wizard Questions 85

6.3 Support not fully compliant IdMPs 87

6.3.1 Security Experts: DB population 88

6.3.2 Developers: Wizard Questions 88

6.4 Implementation and Experimental Analysis 89

6.4.1 mIDAssistant Workflow 90

6.4.2 Security Analysis 91

Chapter 7 mIDAssistant Applications on the Real World Scenarios **94**

7.1 Multiple IdMPs 94

7.1.1 TeamApp Scenario 95

7.1.2 Pull Printing Scenario 97

7.1.3 API Assistant 99

Chapter 8 mIDAssistant iGov Extension **103**

8.1 Wizard-based Approach Modifications 103

8.1.1 DB population 104

8.1.2 Wizard Questions 104

8.2 Implementation and Experimental Analysis 105

8.2.1 mIDAssistant_iGov Plugin 106

8.2.2 Security Analysis 108

8.3 iGov Application on SPID 108

Chapter 9 Related Work **111**

9.1 IdM Plugin Competitors 111

9.2 Approaches for Detecting Implementation Mistakes . . . 112
9.3 Threat Modeling Research Papers . . . 112
9.4 Wrong AD/SSO Implementations Detection Tools . . . 112
9.5 Privacy-Preserving Solutions . . . 115

Chapter 10 Conclusion and Future Work **117**
10.1 Future Work . . . 118

Bibliography **120**

Appendix A Downloaded Apps and Automatic Selection S1. **132**
A.1 Details . . . 132
A.2 Accuracy . . . 133
A.2.1 False Negative *C* Apps in Download and S1. . . . 133
A.2.2 False Positive *C* Apps in S1 (detected in S2.) . . . 133

Appendix B Static Analysis Tools **135**
B.1 JADX . . . 135
B.2 SUPER Android Analyzer . . . 137

Appendix C Dynamic Analysis Tools **138**
C.1 Non-Commercial Dynamic Analysis Tools . . . 138
C.1.1 DROZER . . . 138
C.1.2 DroidBox . . . 138
C.1.3 Inspeckage . . . 139
C.1.4 CuckooDroid . . . 139
C.1.5 StaDynA . . . 139
C.1.6 MobSF . . . 139
C.2 Commercial Dynamic Analysis Tools . . . 140
C.2.1 AppKnox . . . 140

C.2.2 AppCritique . . . 140
C.2.3 IBM Application Security . . . 140
C.2.4 ImmuniWeb . . . 141
C.2.5 NVISO . . . 141
C.2.6 TraceDroid . . . 141
C.2.7 AppWatch . . . 141
C.2.8 Nowsecure . . . 142
C.2.9 App-Ray . . . 142
C.2.10 Approver . . . 142

Appendix D Popular OAuth/OIDC IdMPs Analysis Result (2019) 143

D.1 Discussion on *R1*. . . . 143
D.2 Discussion on *R2*. . . . 145

Appendix E Top-Ranked Google Play Store Compliance Analysis Results (2019) 146

E.1 Discussion on *R1* . . . 146
E.2 Discussion on *R2* . . . 147
E.3 Discussion on *R3* . . . 149
E.4 Discussion on *R4* . . . 149

List of Figures

1.1 Native App developer's Challenges for implementing *Graphy* Example. 14

2.1 OAuth Flows. 23
2.2 Code Interception Attack. Image source `https://tools.ietf.org/html/rfc7636` 26

3.1 Relation among Attacks, Threats, and BCPs. 32
3.2 Authorization Code Flow with PKCE for Native Apps. 35
3.3 The Relation among OAuth, OIDC, iGov, and FAPI profiles. 40
3.4 BCPs-Compliant Request Schema. 53

5.1 An Excerpt Risk Model, Related to Feature *Code Challenge Method*. 82

6.1 Approach. 84
6.2 mIDAssistant Plugin Workflow. 89

7.1 TeamApp Screen Shots. 95
7.2 The Comparison between Current and Proposed Approach. 96
7.3 mIDAssistant GUI. 98
7.4 The High-Level Architecture of STAnD Project. 99
7.5 LibPecker Third-Party Library Detection Tool Architecture. 100
7.6 STAnD Identity Management Plugin GUI. 101

8.1 mIDAssistant iGov Plugin Workflow. 106

8.2 Proposed Approach to Support iGov (SPID Scenario). 109

List of Tables

3.1 OIDC query parameters of Authorization Code Flow with *PKCE* for Native Apps. 36
3.2 OAuth/OIDC Security BCPs for Native Apps. 38
3.3 Threats, OAuth/OIDC Security BCPs countermeasures, and Attacks. 39
3.4 iGov Scenario. 41
3.5 `client_assertion` JWT Structure. 42
3.6 iGov Security BCPs for Native Apps. 43
3.7 Threats, iGov Security BCPs countermeasures, and Attacks. 44
3.8 FAPI Security BCPs for Native Apps. 46
3.9 Threats, FAPI Security BCPs countermeasures, and Attacks. 46
3.10 OAuth/OIDC core, iGov, FAPI profiles Security BCPs Comparison. 47
3.11 Privacy BCPs. 51

4.1 IdMPs Status. 57
4.2 IdMPs Compatibility with AppAuth SDK. 60
4.3 Automatic and Manual selection (S1.-S2.), Static Analysis (S3.), Reaching 10 Apps (S4.) and Resulting Datasets (RD 2019, RD 2021). 62
4.4 Overview of the Selected Features per IdMP. 65
4.5 Security BCPs Coverage per IdMP. 67
4.6 Overlooked security BCPs per IdMP by native app developers. 68

5.1 OAuth/OIDC Reference Model. 74

5.2 Atomic Features Definitions. 78
5.3 Composed Features. 79
5.4 List of Known IdMPs and their Supported Features. 81

A.1 Details about Downloaded Apps and Automatic Selection S1. 132

B.1 Static Analysis Tools Overview. 136

D.1 IdMPs Status 2019. 144

E.1 Automatic and Manual selection (S1.-S2.), Static Analysis (S3.), Reaching 10 Apps (S4.) and Resulting Datasets (RD 2019, RD 2021). 147
E.2 Security BCPs coverage per IdMP. 148

Chapter 1

Introduction

1.1 Context and Motivations

***Mobile Digital Identity*.** In recent years, the rapid development of mobile technologies has turned these devices from entertainment gadgets to popular and ever-present media. Nowadays, users can manage their banking accounts, control their personal health records, or access public administration services by installing a mobile native application (hereafter, native app or simply app when it is clear from context). Although that provides benefits to both service providers and users by allowing them to perform online what they were required to execute in person, it comes with new security and privacy issues, such as the loss of personal data. Thus, protecting sensitive data has become of paramount importance.

Given that, a widely adopted approach is to distrust any entity until it is authenticated at an appropriate level of assurance and has proven to hold proper permissions to access resources. As a result, mobile digital identity becomes the basic building block on top of which to develop secure and trustworthy services and web/mobile native applications in several domains such as public administration, healthcare, and finance. More precisely, authentication and authorization are prerequisites of any access to or operation on resources. To illustrate, let us consider a platform like IFTTT[1] that allows for connecting native apps, devices and services from different native app developers in order to trigger one or more automations. For instance, the Qapital native app for personal finance[2] is designed to motivate users to save money through gamification of their spending behavior. Qapital has been integrated with IFTTT to permit the combination with other services (such as Fitbit or Uber) in order to associate positive behaviors (e.g., perform a certain amount of daily workout or prefer an Uber over a personal car, respectively) and push even further the idea of gamification of the spending behavior. To support this kind of integration, the

[1] `https://ifttt.com/`

[2] `https://www.qapital.com/`

IFTTT platform gets privileged access to user's online services (and smart devices) by using the OAuth 2.0 (OAuth) protocol [Har12] for access delegation. Another example is *Wish*, a famous shopping native app that allows users to log in by either username and password (traditional log in) or use the Facebook login button that manages the user authentication by providing a single sign-on experience. While for the former, native app developers need to store and manage the user credentials, for the latter, they delegate the user authentication task to Facebook. To integrate single sign-on login, *Wish* uses the OpenID Connect (OIDC) protocol [SBJ+14] to authenticate users without the need for storing and managing the credentials of the users where the accounts are located while permitting access to the data necessary to perform the login. Note that, in order to assist native app developers with the implementation of the previous examples, IFTTT and Facebook may provide SDKs that enable Qapital and *Wish* apps to authorize and authenticate users by communicating with the IFTTT and Facebook, respectively.

As pointed out in the previous examples, the following two are the most frequently used scenarios in which APIs for Identity Management (hereafter, IdM) are made available by third-party IdM Providers (IdMPs), such as Google, OKTA, and Auth0 (to name but a few):

Access Delegation (AD). It is used when an web/mobile native applications need to access protected APIs (pointing to resources of the user) on behalf of the user through an IdMP. This requires explicit authorization from the user. Considering the IFTTT example, Qapital is playing the role of the service provider app, user's financial information is the protected resource, and the banks that are hosting the user's financial information play the role of IdMPs (a.k.a. authorization servers).

Single Sign-On (SSO) Login. It is widely used by web/mobile native application developers to allow users to use the same credentials along with different web/mobile native applications, by requiring a single registration process on an IdMP. This is helpful both for usability and security purposes. Considering the shopping native app scenario, *Wish* is playing the role of the service provider app and Facebook the role of IdMP. This enables users to use their Facebook credentials to log in within *Wish*.

OAuth and OIDC Universe. OAuth [Har12] and OIDC [SBJ+14] are two of the most widely used protocols to support digital IdM in innovative web/mobile native applications, by supporting the AD and SSO Login scenarios, respectively. They are provided by the OAuth Working Group (WG) and the OpenID Foundation (OIDF), respectively.

At the beginning, the OAuth world was quite simple. It was composed of a core document that explains how to obtain an access token from the authorization server [Har12], a document to define how to use the aforementioned access token [JH12], and finally a complementary document [LMH13] that explains the OAuth 2.0 Threat Model and Security Considerations. However, after a while, many other documents have been published by the OAuth WG [DB17, JBS15, CMJG15, JCM15, JBM15, JNC+20, SBA15] to introduce new mechanisms

and suitable solutions for the various requirements that are risen due to the broader application of OAuth. In addition, they have introduced various profiles on top of OAuth/OIDC core that inherits all properties of the OAuth/OIDC core standards and aim to increase the baseline security and interoperability by providing specific implementation guidelines for use case scenarios that demands a higher security level (e.g., Finance). Some good representative of such profiles are "International Government Assurance (iGov)", "Financial Grade API (FAPI)", and "OpenID Connect for Identity Assurance profiles" [ID18a, ID18b, SBJ21a, SBJ21b, LF20]. Given that, the related OAuth/OIDC information is becoming very sparse as the information is distributed within various documents. As a result, their secure integration within web/mobile native applications is becoming complex as web/mobile native application developers need to read many documents and understand the fundamentals and security-related concepts within these documents, which is not an easy task for non-security experts. This can be even more difficult for the native app developers due to the peculiarities of native apps.

The aforementioned complexity, whose burden is on native app developers shoulder, has led to various implementation flaws in integrating OAuth and OIDC solutions within native apps. Wang et al. [WZL+15] analyzed the OAuth implementation of more than 4100 popular Chinese apps and their analysis revealed that 86% are vulnerable. The OAuth implementation analysis of Rahat et al. [ARFT19] reveals that, out of 600 apps under their investigation, 32% of apps are vulnerable. A recent example of the wrong implementation of OAuth in a famous Mobile Banking app is reported in [Ker21]. The app implemented the *Implicit* flow–that is a not recommended OAuth flow–that led to account takeover of 1 million customers.

To assist native app developers to easily access information in an intelligible and plain language, in this thesis, we demystify the OAuth/OIDC core documentations and two of the most security critical profiles, namely iGov and FAPI, and summarize the wealth information in plain English.

OAuth Best Current Practices and their Adoption. To secure the integration of OAuth/OIDC solutions in the wild, the OAuth WG and the OIDF have published many security-related documents, providing general guidelines and best current practices (BCPs) for OAuth/OIDC implementers [LMH13, LBLF21, Har12, SBJ+14]. BCPs are defined as a proper mechanism for minimizing the impact of attacks on web and mobile native applications, and they present a reliable and tested solution to deal with recurring security threats [OMGMR+10]. These documents provide specific recommendations and easy to implement mitigations for both IdMPs and web/mobile native application developers that aim to avoid from being attacked through implementation weakness and known anti-patterns and to give additional protection to scenarios that demand a higher level of security, such as Fintech. In the case of IdMPs, these documents explain the recommended features that IdMPs must support to achieve the desired level of security. For web and mobile native application developers, they define how OAuth/OIDC solutions should be implemented for the specific use case scenario to avoid possible threats. In addition, due to the peculiarities of native apps, the OAuth WG has published the "OAuth 2.0 for Native Apps" documentation dedicated to assist native app developers. Recently, the OAuth WG made a fur-

ther effort to support native app developers by releasing AppAuth, a client SDK for native apps to communicate with IdMPs by following the security and usability BCPs described in [DB17]. To attest the applicability and effectiveness of the BCPs, [FKS16, FKS17, Hof19] have formally proved the mitigation of a wide range of threats in the presence of recommended BCPs.

To verify the applicability and the role of the BCPs described in [DB17] to increase the security of IdMP solutions and consequently their effect on the secure integration of IdMPs' solution within native apps, in this thesis, we perform a survey on IdMPs and top-ranked Google Play Store apps to check their compliance against the aforementioned BCPs. To this aim, we define a selection procedure for popular IdMPs and relevant top-ranked Google Play Store apps and specify and extract the features for selected IdMPs besides Google Play Store apps to check their compliance concerning the BCPs.

We selected and analyzed 14 popular IdMPs to check (i) how many IdMPs are violating BCPs, and (ii) how many IdMPs are compatible with AppAuth SDK. While for the top-rank Google Play Store apps analysis, we analyze 80+ apps by selecting them out of 2505 apps from 13 varied categories to check (i) how many apps are violating BCPs for native apps (e.g., by using the *Implicit* flow or an embedded browser); (ii) how many apps contain a flawed implementation (e.g., a misconfiguration of HTTPS scheme or a secret saved in plaintext); and (iii) how many apps that are using official IdMP SDKs modify them. Interestingly, despite the efforts of the OAuth WG and OIDF in publishing BCPs to improve the security of IdM solutions provided by IdMPs and increase the security awareness of native app developers, our analysis shows that: (i) several IdMPs still do not follow BCPs and their documentation is usually unclear or misleading for the native app developers (e.g., sometimes they do not even mention which flow must be used in case of native apps), (ii) many apps under analysis still do not follow the BCPs as native app developers are using known anti-patterns, such as the *Implicit* grant flow. This issue can be due to the wrong interpretation of OAuth/OIDC documentation by native app developers that most of the time are not security experts and give precedence to provide new functionalities without considering security, and (iii) none of the apps under analysis use AppAuth SDK.

Motivating Example. For the sake of concreteness, we consider a scenario (inspired by [Sak]), in which a native app (let us call it *Graphy*) gets the user's financial data from its online bank accounts and makes a graphical representation of their financial status. This scenario includes both the SSO login and AD scenarios, as commonly happens in real-world apps. To access the financial services on *Graphy*, the user should log in to *Graphy* by using either traditional login methods or SSO login solutions that are integrated in *Graphy* (e.g., OKTA or Google). After the authentication, the user can connect its online bank accounts to the app. To do so, *Graphy* implements an AD solution that enables users to authorize the app to access their financial information on their behalf and aggregate their account balances in one place. Finally, *Graphy* provides a graphical representation of users financial status by using their financial information.

The effort required to implement *Graphy* is substantial. As shown in Figure 1.1, a native app developer should:

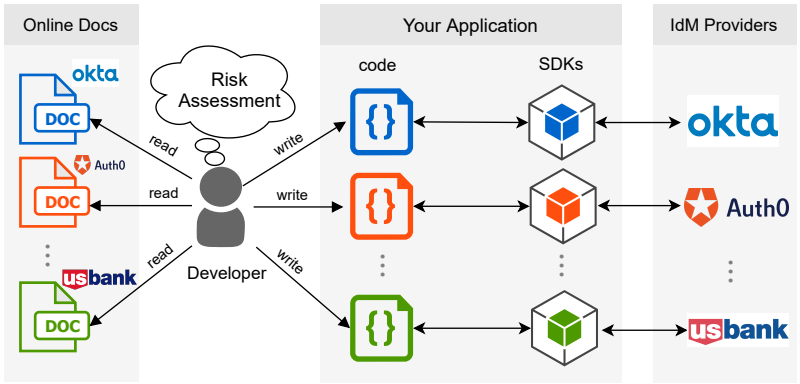


Figure 1.1: Native App developer's Challenges for implementing *Graphy* Example.

- Refer to the documentation of each employed IdMP s/he plans to integrate within the app.
- Refer to the varied OAuth/OIDC documentations and their related security implications to understand the basic concepts and identify the BCPs that should be integrated within the app to secure their OAuth/OIDC solution.
- Perform a risk assessment procedure to estimate the risks posed by varied implementation choices and make an informed decision aiming at minimizing the risk.
- Download the different SDKs of each employed IdMP and understand how to interact with them to integrate the IdMP solution.
- Write different pieces of code to integrate the SDKs to enable the app communication with employed IdMPs.

Summary of secure OAuth/OIDC implementations challenges. To summarize, we conjecture the following issues concerning the current situation of securing the OAuth/OIDC solutions implementations:

- The BCPs documents are sparse, and they have been published gradually over the years by the OAuth WG. Thus, it is possible that native app developers are not aware of them.
- Native app developers may interpret the OAuth/OIDC documents wrongly as they are not security expert.
- Lack of BCPs adoption by IdMPs may lead to difficulty in the integration of AppAuth SDK within native apps. In addition, native app developers need to write a secure code to invoke the SDK within their native apps properly, which is a daunting task.

- Native app developers are usually under pressure to provide the new functionalities for the native app, and performing a risk assessment procedure is a complex task in the context of OAuth/OIDC solutions.

Note that this is not the case for IdMPs, as they usually have enough resources to perform a risk assessment procedure. The risk assessment enables IdMPs to observe the risks posed by their implementations of alternative security measures in comparison to the ones recommended by the BCPs. Thus, we conjecture that they might be aware of these BCPs violations, and their non-compliant solutions can be due to backward compatibility reasons.

1.2 Contributions

The above-mentioned problems motivate us to propose methodologies to assist native app developers with secure OAuth/OIDC implementations. To this aim, we provide an OAuth/OIDC reference model to assist them in conducting a risk assessment. It enables native app developers to observe the risks posed by varied implementation choices and, more importantly, to learn how to decrease the risk level by changing their implementation choices. In addition, we design a wizard-based approach and implement our approach within an Android Studio plugin called mIDAssistant that automatically enforces BCPs within native apps by leveraging AppAuth SDK. Thus, native app developers do not need to worry about neither the core functionalities related to the OAuth and/or OIDC solutions nor the BCPs enforcement—as our wizard-based approach takes care of them.

More in detail, the contributions are the following:

1. Demystify the OAuth/OIDC core documents and two of the most security critical profiles, namely iGov and FAPI to (Chapter 3):
    a. Extract the wealth information and summarize them in plain English as it is hard to digest by non-security experts;
    b. Identify the security BCPs for the core, iGov and FAPI profiles.
2. Perform a compliance analysis w.r.t. OAuth/OIDC BCPs for native apps (Chapter 4):
    a. On 14 popular IdMPs;
    b. On 80+ top-ranked Google Play Store apps, selected out of 2505.
3. Propose a reference model for the OAuth/OIDC solutions to (Chapter 5):

a. Solve the complexity of performing OAuth/OIDC risk assessment due to: (i) many choices of IdMPs and various implementation options for native app developers, (ii) a maze of documents and guidelines to perform a comprehensive and flawless risk assessment, and (iii) the difficulty to be aware of which BCPs to follow that meets the requirements of their native apps.

b. Increase native app developers security awareness by providing a possibility to perform a trade-off risk analysis concerning different implementation choices that consequently helps to minimize the risks within their implementation.

4. Propose a wizard-based approach to assist native app developers with secure integration of the OAuth/OIDC solutions within their native apps and implement it in a working prototype for the Android studio environment (mIDAssistant plugin) (Chapter 6). The effectiveness of our approach has been verified in a number of relevant real-world scenarios, research and innovation projects (e.g., the EIT Digital activity API Assistant) and in the context of industrial collaborations (Poste Italiane, IPZS) (Chapter 7). Furthermore, we had the opportunity to present our work to the OAuth working group experts (during the OAuth Security Workshop), and they have shown interest in our approach.

1.3 Structure of the Thesis

The thesis is structured as follows.

Chapter 2 - Background

This chapter introduces the reader with the basic notions and definitions on which this thesis work is based.

Chapter 3 - Demystify Best Current Practices for Native Apps

In this chapter, we aim to demystify the BCPs for native apps to assist native app developers with secure OAuth/OIDC implementations within their apps. In particular, we first present the main threats affecting the OAuth/OIDC solutions for native apps. Then, we provide a definition for BCPs, and explain how the enforcement of these BCPs for OAuth/OIDC core and its iGov and FAPI profiles can help to mitigate these threats. Furthermore, we explain the *Authorization Code* flow for native apps by introducing the mandatory query parameters in the authorization and token requests for OAuth/OIDC core and its iGov and FAPI profiles. Finally, we introduce our recommended privacy BCPs to improve the users privacy within OAuth/OIDC implementations.

Chapter 4 - Compliance Analysis of IdMPs and Google Play Store Apps with OAuth and OIDC Security BCPs

This chapter presents the experimental results obtained by performing a compliance analysis on popular IdMPs and top-ranked Google Play Store apps. At first, we introduce the selection procedure and the features that are considered for the compliance analysis. Then, we defined a set of research questions to be answered by our analysis and present our BCPs compliance results on 14 popular IdMPs and 80+ top-ranked Google Play Store apps, selected out of 2505 apps.

Chapter 5 - OAuth/OIDC Risk Assessment

In this chapter, we present the reference model for the OAuth/OIDC solutions implementations. In particular, we first elaborate on the importance of performing a risk assessment procedure in the context of OAuth/OIDC solutions and the problem that should be addressed. Then, we present how the obtained information from the OAuth/OIDC related documents and their security implications can be used to build a reference model for OAuth/OIDC solutions implementations. Finally, we formalize the risk assessment problem and explain how native app developers can leverage the reference model to solve this problem.

Chapter 6 - mIDAssistant Plugin

In this chapter, we present our wizard-based approach that is implemented in the mIDAssistant plugin for assisting native app developers with automatic and secure integration of OAuth/OIDC solutions within their native apps. To this aim, we define the main process of our wizard-based approach. Then, we present how we implement it within the mIDAssistant plugin by providing architecture and the implementation details. Finally, a security analysis on the integrated code by the mIDAssistant plugin is given.

Chapter 7 - mIDAssistant Applications on the Real World Scenarios

In this chapter, we present the application of the mIDAssistant plugin on different use cases. At first, we detail each use case scenario and then explain how our plugin is helpful to address the challenges within each use case scenario.

Chapter 8 - mIDAssistant iGov Extension

This chapter explains the extension we did on the preliminary version of mIDAssistant plugin to support the iGov profile. We detailed the main changes in the wizard. Then, we present the plugin architecture and the implementation details. Finally, we provide some insights on how we can adopt the mIDAssistant_iGov plugin to support the scenario illustrated by users access to governmental and public administration services, such as the one provided by the Italian digital identity framework SPID [AGI19b].

Chapter 9 - Related Work

This chapter compares our work to existing publications, considering the following criteria: plugin competitors, detecting implementation flaws, threat modeling research papers, wrong AD/SSO implementation detection tools, and privacy-preserving solutions.

Chapter 10 - Conclusion

In this chapter, we conclude this thesis work and provide some insights on possible future directions.

Some of the material in this manuscript is based on the following research activities, published and under-review papers at international conferences and journal:

- Sharif A., Carbone R., Ranise S., and Sciarretta G., *A Wizard-Based Approach for Secure Code Generation of Single Sign-On and Access Delegation Solutions for Mobile Native Apps*, In Proceedings of the 16th International Joint Conference on e-Business and Telecommunications - Volume 2: SECRYPT, (ICETE 2019) ISBN 978-989-758-378-0, pages 268-275, https://doi.org/10.5220/0007930502680275.

- Sharif A., Carbone R., Sciarretta G., and Ranise S., *Automated and Secure Integration of the OpenID Connect iGov Profile in Mobile Native Applications*, In: Saracino A., Mori P. (eds) Emerging Technologies for Authorization and Authentication. ETAA 2020. Lecture Notes in Computer Science, vol 12515, Springer, https://doi.org/10.1007/978-3-030-64455-0_4.

- Sharif A., Carbone R., Sciarretta G., and Ranise S., *Best Current Practices for OAuth/OIDC Native Apps A Study of their Adoption in Popular Providers and Top-Ranked Android Clients*, Under revision in Journal of Information Security and Applications, Elsevier.

- Dashti S., Sharif A., Carbone R., and Ranise S., *Automated Risk Assessment and Trade-Off Analysis of OpenID Connect and OAuth 2.0 Deployments*, In Proceedings of the 35th Annual IFIP Conference on Data and Application Security and Privacy (DBSec21), https://doi.org/10.1007/978-3-030-81242-3_19

- Ranise S., Carbone R., Sciarretta G., and Sharif A., *STAnD (Security Tools for App Development) is the outcome of EIT Digital's API Assistant Innovation Activity 2018*, https://stfbk.github.io/projects/STAnD.

Chapter 2

Background

In this preliminary chapter we introduce the reader with the basic concepts and definitions on which this work is based.

2.1 Identity Management

Identity Management (IdM)–also called "Identity and Access Management (IAM)"– provides a way to manage any access to or operation on resources. In particular, it is a key component that should be considered within security architecture. Its primary concern is to check the identity of the entities (called authentication) before granting them appropriate permissions to access resources (called authorization). Thus, it becomes the basic building block on top of which to develop secure and trustworthy services and web/mobile native applications.

We always deal with authentication and authorization mechanisms in our daily life. When users enter a username and password or use biometrics (e.g., fingerprint, face detection), the services and web/mobile native applications verify users identity for authentication purposes. After that, an authorization mechanism is implemented by that service or web/mobile native application to define users access level. For example, the regular user should have a different level of access in comparison with administrative users.

Among other IdM solutions, OAuth and OIDC are the most frequently used standards [Har12, SBJ+14] that provide a secure and frictionless user experience. Thus, in the rest of this thesis, we mainly focus on these two de-factos standards for authorization and authentication.

2.2 IdM Standards

In the following, we will describe the two most widespread standards on authorization and authentication, namely OAuth (Section 2.2.1) and OpenID Connect (Section 2.2.2). We will also present the current status concerning the support of these standards to the mobile context (Section 2.2.3).

2.2.1 OAuth 2.0

OAuth [Har12] is an authorization protocol that regulates what an entity is allowed to access. It is introduced by the OAuth WG to solve the problem of sharing user's credentials (username, password) with third-party web/mobile native applications to access user's information hosted on a different server, which is typical in the case of using HTTP basic authentication.

In simple words, OAuth provides a mechanism by which a user, called Resource Owner (*RO*), can delegate access for selected pieces of information contained in a Resource Server (*RS*) to a designated Client (*C*) web/mobile native application, without having to share her credentials with *C*. Indeed, *RO* direct authenticates by using her User Agent (*UA*) with a trusted server, called Authorization Server (*AS*), which issues a token (called `access_token`) to *C* carrying the requested authorization delegation. Finally, *C* uses the `access_token` to access the *RO*'s resource in *RS*.

The OAuth WG defines two client types, confidential and public clients. The difference between these two client types is whether they can be deployed with some credentials, called `client_secret`. In particular, confidential clients can be deployed with a `client_secret`, which is not visible to the users of the application. This is simply the case for web applications, which can store this secret securely on the web application backend. This value is used by *AS* to authenticate *C* web application before releasing an `access_token` and *AS* will know that only the real *C* web application can make the request. However, this is not the case for the public (native) clients, as the user can use debugging tools to extract the secret from the source code. Thus, in the OAuth context that does not consider the usage of the secure element to manage the client credentials, there is just no way to ship a secret within native apps and have it remains secret. This means that the *AS* never really knows if the request comes from the legitimate *C* app or something pretends to be a legitimate app. The main reason for this distinction is since *AS* might have different policies, which makes it acts differently depending on the type of client making the request. For example, based on the client type *AS* might decide to change the `access_token` lifetime.

The OAuth WG also defines the following four grant types, which are dependent on the involved entities and the relative scenario assumptions: *RO Password Credentials*, *Implicit*, *Authorization Code*, and *Client Credentials*. The aforementioned grant types have a common goal that is re-

leasing an access_token to *C* for granting access to resources; differences among them concern the process to obtain tokens. Here, we provide a brief explanation for each flow.

Client Credentials Flow. The *Client Credentials* flow is used when the authorization is limited to resources under the control of *C* (e.g., during server to server communication). This flow, illustrated in Figure 2.1.a, consists of two steps: In Step 1, *C* requests an access_token and, in Step 2, *AS* authenticates *C* to verify the authenticity of the request, and if valid, issues an access_token.

RO Password Credential Flow. The *RO Password Credential* flow is suitable when there is a high-level trust establishment between *RO* and *C* (e.g., *C* is part of the OS or a high privileged web/mobile native applications). This flow, illustrated in Figure 2.1.b, contains the following steps: In Step 1: *RO* provides *C* with his credentials, namely username and password; After that, *C* uses the received credentials to request an access_token from *AS*. Finally, in Step 3, *AS* validates *RO* credentials, and if valid, issues an access_token. However, the usage of this flow is discouraged by the OAuth WG due to the following reasons: (i) *AS* does not have any way to distinguish between the user's presence behind the flow or a request replay by the web/mobile native application, which is because of the fact that request looks the same, whether the user is actively using the web/mobile native application; (ii) *AS* cannot be sure that users agree with the requested access permissions by the web/mobile native application as there is no consent screen in the flow; and (iii) there is no room in the flow to add multi-factor authentication unless web/mobile native application developers do it on their own and add it to their web/mobile native applications.

Implicit Flow. The *Implicit* flow is optimized for *UA*-based apps, which are developed using a scripting language. This flow, illustrated in Figure 2.1.c, encompasses of the following steps: in Step 1, *C* initiates the flow by redirecting the *RO*'s *UA* to *AS*. *C* also includes a redirection URI (redirect_uri), which determines to which *AS* will send *UA* back once access is granted (or denied). Then, *AS* authenticates *RO* by displaying the login form (within *UA*) that asks for *RO*'s credentials and establishes whether *RO* has granted or denied *C*'s access request. Assuming *RO* has granted the access, *AS* redirects *UA* back to *C* (Step 3), which contains an access_token within the redirect_uri fragment. In Step 4, *UA* makes a request (which does not include the fragment) to the Web-Hosted Client Resource (*WHCR*) that is a web server or a local resource accessible to *UA* and capable of extracting the access token. Note that, the fragment retained in *UA* locally. In Step 5, *WHCR* returns a web page capable of accessing the full redirect_uri including the fragment retained by *UA*. After that, *UA* uses the provided script by *WHCR* to extract the access_token. Finally, in Step 7, *UA* passes the access_token. However, the OAuth WG recommended not to use the *Implicit* flow as it is vulnerable to access_token leakage and access_token replay as detailed in the OAuth BCPs [LBLF21].

Authorization Code Flow. The *Authorization Code* is the most popular grant type, which is suitable for both confidential and public clients. It is worth mentioning that the *Authorization Code* flow possesses some security benefits in comparison with previous flows, such as the abil-

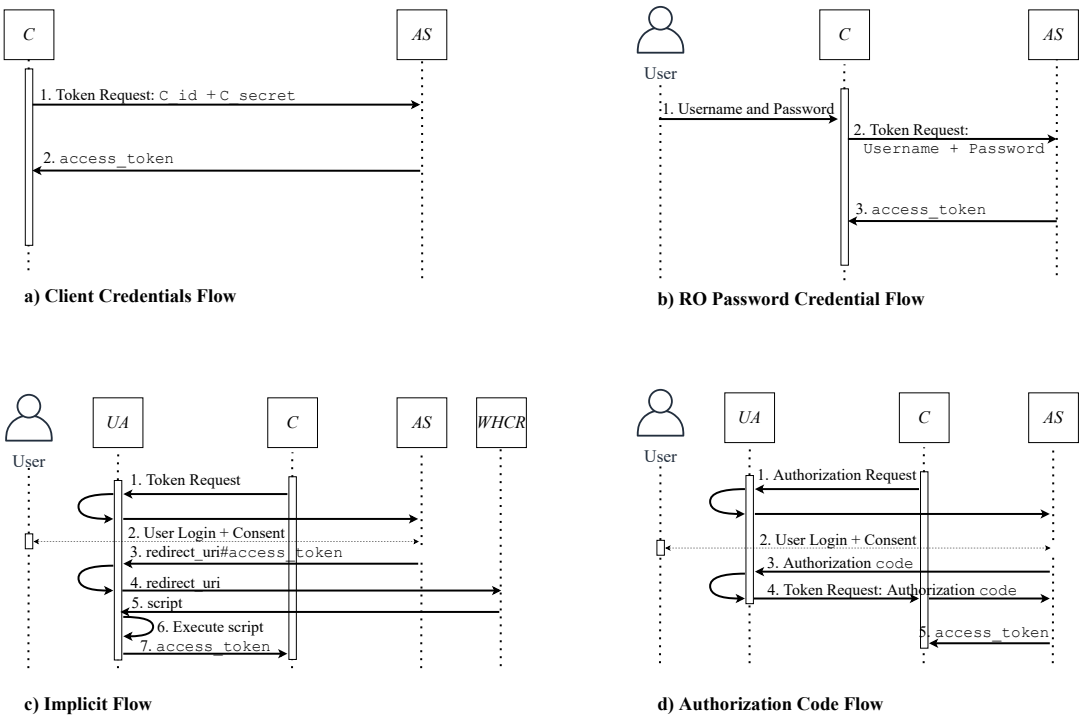


Figure 2.1: OAuth Flows.

ity to authenticate *C* and the direct transmission of the `access_token` to *C* without passing it through *UA*, thus avoiding the exposure of the `access_token` to others, including *RO*. This flow, illustrated in Figure 2.1.d, encompasses the following steps:

- Steps 1-3 are the same as Steps 1-3 in the *Implicit* flow (described before). The difference is that the `redirect_uri` does not include directly the `access_token` and instead it includes an Authorization `code`.
- In Step 4, *C* requests an `access_token` by authenticating at *AS* and presenting the Authorization `code`.
- Finally, in Step 5, *AS* authenticates *C* and validates the authorization code, and if valid, issues an `access_token`.

2.2.2 OpenID Connect

OIDC [SBJ+14] is an authentication layer developed on top of the OAuth standard. The problem that has led to introduce OIDC is the fact that the OAuth protocol is often abused to implement authentication by major web/mobile native applications, while its main purpose is AD.

In [CPC+14], the authors studied several vulnerabilities that arise when using OAuth for authentication. Furthermore, the aforementioned study highlights the main demands that an authentication protocol should satisfy.

OIDC adds two main features into the OAuth standard: the `id_token` and the *userInfo* endpoint. The `id_token` is a structured JSON token [JBS15] that contains information about the token issuer (the OIDC provider), the subject (user identifier) and the audience (the intended *C* web/mobile native application), all signed by the OIDC provider. This token enables *C* to safely verify that the received token is issued as a result of its previous token request. While the *userInfo* endpoint is used to obtain identity-related attributes (e.g., the email and address).

2.2.3 Native Application Scenario

This section explains the differences between web and mobile environment, also known as native app scenario. Our motivation to explain these peculiarities highlights how these differences can affect the OAuth/OIDC solutions security.

Native app scenario in the OAuth context is where a native app plays the role of *C*. As native apps cannot be deployed with client credentials, as mentioned in Section 2.2.1, they are considered public clients. One of the main differences in native app scenario w.r.t. web scenario is the way that mobile platforms like iOS and Android handle the redirection between *AS* and *C*. In the case of web applications, the whole interaction happens within the browser. Thus, it can benefit from all the browser advantages such as DNS and HTTPS checks. However, for native app scenarios, we lose many of those protections built into the browser as the user starts in the app that launches a browser and then *AS* sends a redirect to the app. The OAuth WG does not specify a unique solution for the redirection between *AS* and *C*. They propose the following three different solutions:

App-declared Custom URI scheme. It allows native apps to register with OS a Custom URI scheme (e.g., "com.test.app"). In this way, if the browser or another app tries to load a custom URI scheme, the app that registered it is launched to handle the request.

App-claimed HTTPS URI scheme. Some mobile OS[1] allows the app to register HTTPS URI scheme, which are the domains they control. In this case, whenever the web browser is redirected to the HTTPS URI scheme, the operating system launches the native app.

Loopback URI redirection. This method handles the redirection by creating a local HTTP listener on a random port that receives the redirection request on the determined listener. It is worth mentioning that this method can be used just if native apps can open a port on the loopback interface without special permission (typically for desktop native applications).

[1]This solution is currently available only for Android 6 (and above) and iOS 9 (and above).

In addition, as described in [JCL+17] open an HTTP listener on Android ports can lead to different vulnerabilities. Thus we do not analyze Loopback URI in this work.

Another essential difference is illustrated by the implementation of *UA*. While in the web scenario, system browsers are widely used as *UA*, in the native scenario, there are two kinds to be used as *UA*, namely: embedded (e.g., Web View) and external (e.g., OS browsers app, in-app browser tabs (Custom Tabs) and IdMP native apps). The implementation of *UA* is crucial as the security of native app developers OAuth/OIDC implementations in the native scenario is dependent on how the *C* app launches the browser to start the flow. While previously the most adopted approach was the usage of Web View, it has been replaced later by the introduction of Custom Tabs due to its security advantages. Custom Tabs as defined in [DB17] is "a full page browser with limited navigation capabilities that is displayed inside a host app, but retains the full security properties and authentication state of the system browser". The Custom Tabs has introduced a new mechanism to implement browser-based protocols and consequently, to support native scenarios as reported in [Mad15a, Mad15b].

Finally, it is worth mentioning that the OAuth WG highly recommends the usage of AppAuth SDK [DB17] to enforce the BCPs for the native scenario (described in Section 3.3.1). AppAuth is an open-source library primarily developed by the Google Identity Platform team and maintained by the OpenID foundation since 2016. AppAuth is used within Google's own client-side Authentication stack in Android and reviewed by Google's internal security team for detecting vulnerabilities. It is worth mentioning that as the library's main aim is to act as the communication facilitator, it is not entering the OIDC relying party certification program. Indeed, it is out of the library scope to implement all the optional features for OAuth/OIDC relying party that are needed for the relying party certification program (e.g., JWT Authentication).[2] The current library can be downloaded, for both Android and iOS, at `https://appauth.io`.

2.3 OAuth Extensions

In the following, we will describe two OAuth extensions that are used in the rest of this thesis, namely Proof Key for Code Exchange (Section 2.3.1) and Dynamic Client Registration (Section 2.3.2). The OAuth WG has provided the extensions mentioned above to address the arisen requirements by the broader usage of OAuth solutions in the varied use cases.

[2] Private communication with one of the native app developers of the AppAuth SDK.

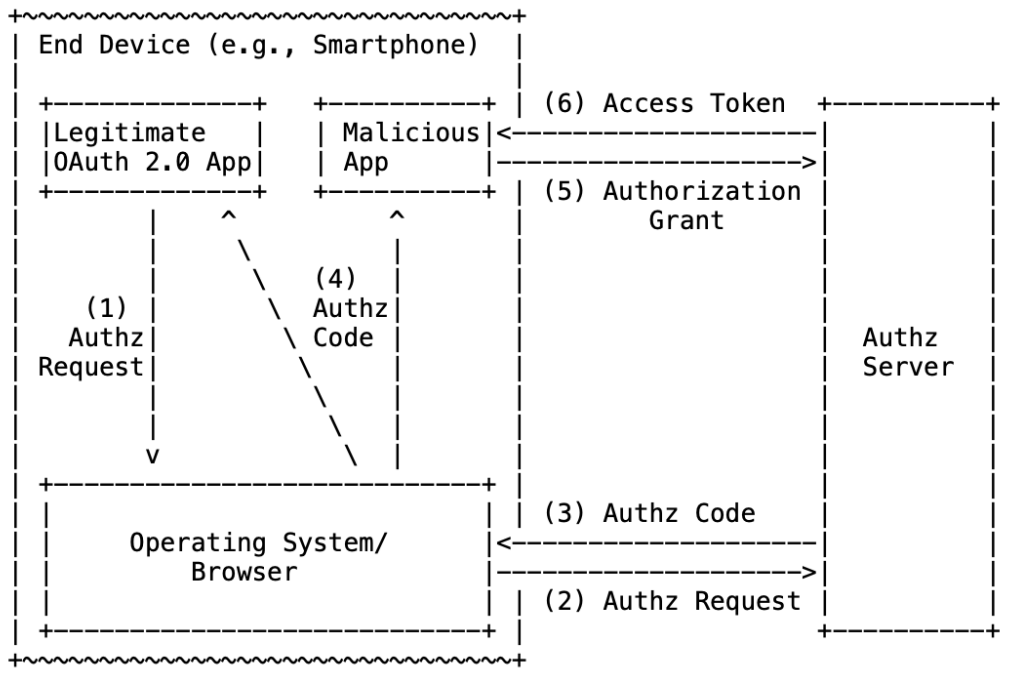


Figure 2.2: Code Interception Attack. Image source `https://tools.ietf.org/html/rfc7636`

2.3.1 Proof Key for Code Exchange (PKCE)

PKCE is short for Proof Key for Code Exchange [SBA15] is an extension to the OAuth core standard, which is originally created to secure the *Authorization Code* flow implementation within public clients against several attacks such as code interception. However, *PKCE* provides security benefits that makes it valuable across all client types [LBLF21].

In simple words, code interception attack refers to scenarios that the attacker can obtain the Authorization `code` in some way and then send it to *AS* to exchange it for an `access_token`. There are a couple of available ways that an attacker may use to steal the code. Figure 2.2 presents a scenario where a malicious native app also registers a custom URI scheme that is similar to the one in the legitimate native app. Thus, the malicious app can steal the code from the authorization response and then exchange it for an `access_token`.

To mitigate this attack, *PKCE* extension introduced three query parameters to provide a way for the app to prove to the *AS* that the Authorization `code` belongs to the app, namely: (i) `code_verifier`, (ii) `code_challenge`, and (iii) `code_challenge_method`. The `code_verifier` is a dynamically created cryptographically random key, which is unique per authorization request. The `code_challenge` is a transformation of the `code_verifier` by using the method that

is declared in code_challenge_method (e.g., Plain or S256). Thus, an attacker needs to know the value of the code_verifier for the intercepted request to be able to exchange the Authorization code for an access_token, which is not as simple as code interception. Note that the following changes will be applied to the *Authorization Code* flow as illustrated in Figure 2.1: In Step 1, the *C* sends together with OAuth query parameters (e.g., client_id, redirect_uri), the *PKCE* query parameters, namely: code_challenge, and code_challenge_method. Steps 2-3 will remain unchanged, and in Step 4, *C* sends the authorization code with code_verifier to *AS*.

2.3.2 Dynamic Client Registration (DCR)

DCR is short for Dynamic Client Registration [JBMH15, SBJJ13] is an extension to the OAuth core standard, which enables *C* native app to obtain dynamically per app instance client identifiers. IdMPs may implement the DCR endpoint as either an OAuth protected or publicly accessible resource as reported in [JBMH15]. While in the former case, the protected DCR endpoint may require an initial access_token that is obtained beforehand during the registration phase by native app developers, in the latter case, it is publicly accessible to native app developers.[3]

Considering the protected DCR scenario, in the following we describe the general DCR flow:

a. The *C* app initiates the flow by setting the authorization header with the previously obtained access_token and sending the client registration request to the DCR endpoint to register a new *C* app with the IdMP. This request contains the following query parameters:

   application_type: defines the application type. In the case the query parameter is not present, IdMP sets the value into web.

   jwks: *C*'s JSON Web Key Set document [Jon15]. Firstly, native app developers programmatically create a pair of public/private key for *C* app. Then, while the private key is stored securely by *C* app (e.g., in the Keystore), the public key is shared with IdMP by using jwks query parameter. IdMP may use *C*'s public key to validate signatures from *C*. jwks must be set to provide support for private_key_jwt authentication.

   token_endpoint_auth_method: defines which client authentication method should be used in the token endpoint. The default value is set into client_secret_basic.

   response_type (Optional): defines the response_type that *C* restricts itself to using. The default value is set to code.

   grant_type (Optional): defines the grant_type that *C* restricts itself to using. The default value sets into authorization_code.

[3] In the case of publicly accessible DCR endpoint, requests may be constrained through various techniques (e.g., rate limited) to prevent DoS attacks.

`redirect_uri`: the URL to return to after the registration.

b. The DCR endpoint processes the request and releases a new `client_id`.

2.4 OAuth and OpenID Connect Profiles

In the following, we will describe some OAuth/OIDC profiles, namely International Government Assurance Profile (Section 2.4.1), Financial Grade API Profile (Section 2.4.2), and OpenID Connect for Identity Assurance Profile (Section 2.4.3).

2.4.1 International Government Assurance Profile (iGov)

Electronic identification schemes have been built to simplify citizen's access to online public administration services and reduce password fatigue via a single sign-on login solution. To provide a precise specification for government and public service domains on protecting the user's identity information and activity from unintentional exposure, the OAuth WG and the OIDF have published the International Government Assurance Profile (iGov) document [ID18a, ID18b].

In simple words, these specifications profile the OAuth/OIDC core to increase the baseline security, provide higher interoperability, and structure deployments in a manner specifically applicable to government and public service domains. In particular, the iGov profile mandates the usage of optional query parameters within OAuth/OIDC core to increase the baseline security to an appropriate level for the government and public domain sectors. To illustrate, let us considered the OIDC `nonce` query parameter. While its usage is optional for the OIDC core implementation, the iGov profile enforces its implementation within the iGov profile as it provides additional countermeasures for mitigating session misuse and obtaining access token (described in Section 3.3.2.2).

2.4.2 Financial Grade API Profile (FAPI)

FAPI is short for Financial Grade API, which is a technical specification developed by the FAPI WG of OIDF. The FAPI profile uses OAuth [Har12] and OIDC [SBJ+14] as its foundation and defines additional requirements for the financial industry and other use cases that demand a higher level of security in comparison to the baseline security provided by the OAuth/OIDC core standards.

The FAPI WG has published two security profiles as follows:

- FAPI-Part1:Read-Only API Security Profile [SBJ21a], which is suitable for read-only API access to financial data and other similar use cases;
- FAPI-Part 2: Read and Write API Security Profile [SBJ21b], which is suitable for reading and write API access to financial services and other similar situations where the risk is higher.

Note that the letter one (FAPI-Part2) provides higher security measures by utilizing advanced features defined in OIDC [SBJ$^+$14] in addition to the OAuth [Har12]. In the following, I summarize very briefly the enhancements that FAPI-Part2 provides:

- provides authorization request and response message authentication and integrity by using request object (`request`) [SBJ$^+$14], and JWT Secured Authorization Response Mode (`JARM`) [LC18a]. While the former provides message authentication and integrity by enabling the OAuth/OIDC request to pass in a single, self-contained query parameter and be optionally signed and/or encrypted, the letter secures the OAuth/OIDC response by providing a mechanism to encode OAuth/OIDC authorization response query parameters. In particular, all the response query parameters are conveyed in a JSON Web Token (JWT) that can be either signed or signed and encrypted [LC18b].
- prevents client impersonation by adopting asymmetric client authentication methods namely: `mutual TLS`, `client_secret_jwt`, and `private_key_jwt` [CBSL19, SBJ$^+$14];
- prevents unauthorized use of `access_token` by adopting token binding approaches such as token binding and certificate-bound `access_token` [JCBD18, CBSL19].

2.4.3 OpenID Connect for Identity Assurance Profile

The OIDC for identity assurance specification [LF20] is an extension to OIDC core [SBJ$^+$14] for providing clients with identity information, i.e., verified claims and the verification evidence which are used to verify the claims of the user. Thus, it provides a generic mechanism for clients that avoids a mixture of verified and unverified claims.

This profile is specifically designed for the use cases that demand strong identity assurance to comply with potential regulatory requirements such as Anti-Money Laundering laws. In such cases, clients' main concerns are trustworthiness or assurance level of the claim about the user that an Identity Provider is willing to communicate and not the level of assurance that is asked by an Identity Provider during the authentication transaction (i.e., if 2FA applied). Thus, the `acr` claim introduced in [SBJ$^+$14] is not suited for this kind of use case, and this profile defines a new representation to convey the assurance data about the user.

In order to support the provision of verified identities the OIDC core standard is extended in three places:

- the optional `claim` query parameter in the OIDC core [SBJ+14] is extended to support `verified_claims` element;
- `verified_claims` element is defined for the `id_token` to return the actual verified claims and metadata about the verification.
- `txn` claim as defined in [HJDA18] is used to build audit trails across the parties involved in the OIDC.

Chapter 3

Demystify Best Current Practices for Native Apps

In this chapter, we aim to demystify the BCPs for native apps to assist native app developers with secure OAuth/OIDC implementations within their native apps. To this aim, we (i) present a list of threats concerning OAuth/OIDC implementations within native apps to inform native app developers about the possible impact of wrong implementation choices, (ii) extract the wealth information from varied OAuth/OIDC documentations to provide all the necessary implementation details for the suggested flow within native apps and summarize them in play English as they are hard to digest by native app developers who are not (necessarily) security experts, and (iii) identify the BCPs that should be integrated by native app developers to provide a countermeasure against the identified threats.[1]

3.1 Context and Motivations

The OAuth WG and the OIDF have published a set of BCPs to assist native app developers with the secure integration of OAuth/OIDC solutions (see Section 1.1). To illustrate the importance of the BCPs and how these BCPs can help to mitigate possible threats, we present the relation among threats, attacks, and BCPs as shown in Figure 3.1. The threats are manifested as attacks, and they will have an impact on the user's resources if an adversary can successfully launch them against native app developer's app. However, as it is represented in Figure 3.1 and we will discuss later, the BCPs can help to mitigate threats, attacks and thus secure the user's resources. As such, it is important to make native app developers aware of the related BCPs concerning OAuth/OIDC solutions for native apps.

[1]This chapter is based on our under revision work in [SCSR21].

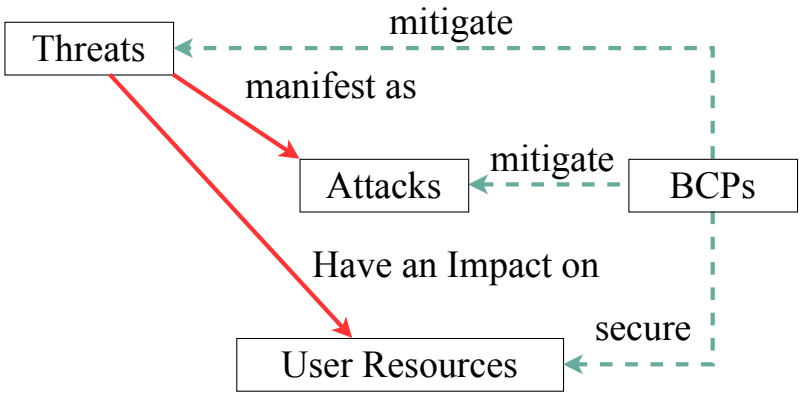


Figure 3.1: Relation among Attacks, Threats, and BCPs.

To this end, in the following, we first present the main security threats affecting the implementations of OAuth/OIDC solutions for native apps in Section 3.2. Section 3.3 provides a definition of BCPs and explains how the enforcement of these BCPs for OAuth/OIDC core (Section 3.3.1), iGov (Section 3.3.2), and FAPI profiles (Section 3.3.3) help to mitigate these threats. Furthermore, we explain the *Authorization Code* flow for native apps by introducing the mandatory query parameters in the authorization and token requests for OAuth/OIDC core and its iGov and FAPI profiles. Finally, we list of our easy-to-implement privacy recommendations to address the threats related to the identified privacy goals (Section 3.3.4). In this way, native app developers can address the privacy requirement of users.

3.2 Threat Model

To provide the threat model that we consider for the native app scenario, we have identified threats that can be mitigated using the defined BCPs for native apps, reported in the "OAuth for Native Apps" document [DB17]. To identify the threats we studied the following documents, namely the OAuth Authorization Framework [Har12], the OAuth Threat Model and Security Considerations [LMH13], the OAuth for Native Apps [DB17], and the OAuth Security Best Current Practice [LBLF21][2]. The main reasons to consider the aforementioned documents are (i) they have been published by the OAuth WG, who are responsible for the OAuth standard. OAuth WG regularly updates these documents to satisfy the security requirements arising from the broader application of OAuth by extending and updating the related threat model. These updates cover the novel attack scenarios which are used in the wild to attack OAuth implementations through implementations weaknesses and anti-patterns. Thus, they contain the most recent threats and related attacks concerning the OAuth implementations, and consequently, the OIDC

[2]"OAuth Security BCP draft 18" is the last available document during the writing of this work

ones built on top of OAuth, and (ii) the OAuth WG has a close relation with industries which play the key roles in the field of IdM together with researchers. This is also witnessed by the yearly event known as OAuth Security Workshop (OSW) that aims to provide a direct exchange of views among academic researchers, OAuth WG members, and industry to highlight possible flaws, which are left unseen in the initial protocol design. Indeed, further analysis by academia and industry can contribute to improving the security of OAuth by revealing possible security flaws that are not considered in the initial document, and spotted by performing depth formal analysis on the protocol or by introducing novel attack patterns through a new attacker model that is not considered by the initial documentation. The following research works are a good representative for spotting security flaws through formal analysis and/or introducing novel attack patterns [FKS16, FKS17, Hof19, MMSW17].

It is worth mentioning that the definition of threat in the OAuth context is more specific than the standard definition by the National Institute of Standards and Technology (NIST). While NIST [RKJ06] defines a threat as "any circumstance or event with the potential to cause harm to an information system in the form of destruction, disclosure, adverse modification of data, and/or denial of service;" in the OAuth context [LMH13], threat includes only intentional attacks on OAuth tokens and resources protected by these tokens. In this work, focusing on the OAuth security, we followed the definition provided by the OAuth WG.

For each threat, we point out the potential impact caused by the threat, namely the security leakages that an attacker would be able to perform as a consequence of the threat.

- Obtaining Access Tokens (T_{token}, § 4.6 in [LBLF21]). The impact of revealing an `access_token` to an attacker can be severe as, if no additional mechanism is in place to authenticate the client, "the attacker can access all resources associated with the token and its scope [LMH13]." A relevant example of an attack leading to an `access_token` leakage is the Instagram broken authentication, due to the usage of *Implicit* flow [CPT+16]. The *Implicit* flow causes the IdMP to issue the `access_token` directly in the authorization response, which makes it vulnerable to token leakage and replay attacks [LBLF21].

- Obtaining Authorization Code (T_{code}, § 4.5 in [LBLF21]). The impact of revealing an Authorization `code` to an attacker is as follows. An attacker can exchange the Authorization `code` for obtaining the `access_token` and access the user resources associated with it (as done in T_{token}). For obtaining the Authorization `code` an attacker might try to eavesdrop the transmission of the Authorization `code` itself between *AS* and *C*. An example of the code eavesdropping attack has been detected in the Airbnb app as it was using HTTP rather than HTTPS to redirect the Authorization `code` from the IdMP to the Airbnb app. Therefore, an attacker could obtain the Authorization `code` by intercepting the traffic through a MitM proxy and substituting the stolen Authorization `code` with the one in his login request within a legitimate Airbnb app to log in as the victim. The attack mentioned above was presented during the Insomni'hack 2019 event [NS19].

- Obtaining User Credentials (T_{cred}, § 8.12 in [DB17]). Revealing the user credentials to an attacker has a severe impact to allow the attacker to impersonate the user. The usage of an embedded browser in the *C* app can lead to the leak of the user credentials. In particular, if the *C* app is malicious (or compromised), the *C* app can directly misuse its full control ability on the user data, ensured by the embedded browser, to steal the user's credentials. In addition, if the *C* app misconfigures the embedded browser—for instance, if the Java Script capability is enabled—an attacker can inject malicious code on the embedded browser to steal user credentials. An example of this attack was presented in the Black Hat 2016 event [CPT+16] which explained how an attacker that implements an OAuth/OIDC solution inside the embedded browser could misuse a supported feature in the embedded browser to obtain the cookie after user authentication within the app and login into the victim's user account.

- Obtaining a Client Secret (T_{secret}, § 8.5 in [DB17]). In case the `client_secret` is used to authenticate the *C* app, as in the case of confidential web applications, the attacker can impersonate a benign client and use it to access the user resources. Two ways to obtain the `client_secret` from the *C* app are: obtaining the secret from the source code or binary using static analysis and intercepting the communication between the *C* app and the *AS*. An example of an attack to obtain the `client_secret` from the *C* app was presented during the Black Hat 2016 event [CPT+16]. The attack was performed against Twitter OAuth Login using Quora and Pinterest apps by a simple search for "secret" string within the *C* app source code.

- Session misuse ($T_{session}$, § 8.9 in [DB17]). Lack of a mechanism in the authorization request to link client requests and responses enables the attacker to exploit inter-app communication channels to perform a so called Login CSRF [BJM08]: a Cross Site Request Forgery (CSRF) style attack, leading a victim to (unwillingly) access the resources on behalf of the attacker.

3.3 Best Current Practices Definition

As mentioned in Section 1.1, the complexity of the OAuth world has led to various implementation flaws in the secure integration of the OAuth/OIDC solutions in the wild. Thus, the OAuth WG and the OIDF have published a set of BCPs in [DB17, LBLF21, ID18b, ID18a, SBJ21a, SBJ21b] to assist web/mobile native application developers with securing OAuth implementations, and consequently, the OIDC ones built on top of OAuth.

BCPs are defined as a proper mechanism for minimizing the impact of attacks on web/mobile native applications. They present a reliable and tested solution to deal with recurring security threats [OMGMR+10] and to attest the applicability and effectiveness of the BCPs, [FKS16,

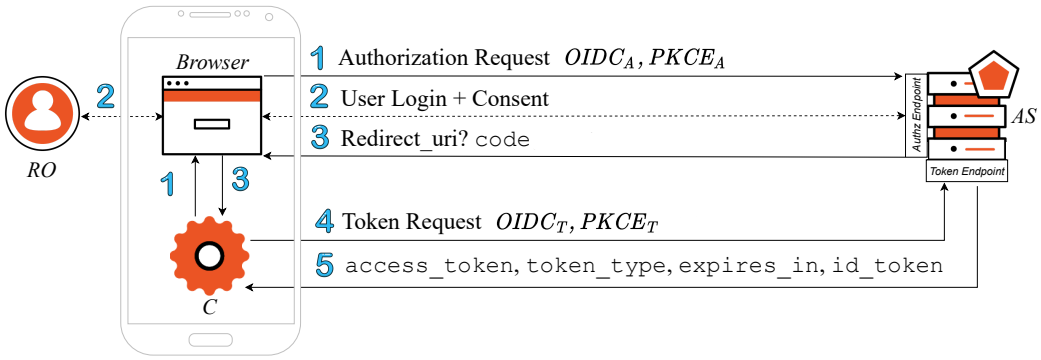


Figure 3.2: Authorization Code Flow with PKCE for Native Apps.

FKS17, Hof19] have formally proved the mitigation of a wide range of threats in the presence of recommended BCPs. These documents provide specific recommendations and easy to implement mitigations for both IdMPs and web/mobile native application developers that aim to avoid being attacked through implementation flaws and known anti-patterns; and give additional protection to scenarios that demand a higher level of security, such as Fintech. In the case of IdMPs, these documents explain the recommended features that they must support to achieve the desired level of security. While for web/mobile native application developers, they define how OAuth/OIDC solutions should be implemented for the specific use case scenario to avoid possible threats. In this thesis, we considered two types of BCPs, namely: security BCPs and privacy BCPs. While security BCPs are extracted from OAuth security-related documents and aim to protect the user resources (e.g., the confidentiality of personal data), privacy BCPs are our specific recommendations to meet privacy goals, such as *Data unlinkability*, and *Data minimization*.

In the following, we summarize firstly the security BCPs for native apps for OAuth/OIDC Core, iGov and FAPI profiles, and then we explain our recommended privacy BCPs.

3.3.1 OAuth/OIDC Security Best Current Practices

As we mentioned in Section 2.2.3, the native scenario contains peculiarities in comparison to the web scenario that could affect the OAuth/OIDC solutions security. To address these peculiarities, the OAuth WG has released the "OAuth 2.0 for Native Apps" [DB17] to provide a set of security BCPs specifically for native apps.

In the following, before elaborating on the security BCPs for native apps, we explain the *Authorization Code* flow for native apps to detail the suggested flow in Section 3.3.1.1. Then, Section 3.3.1.2 describes the OAuth security BCPs for native apps and explains how the enforcement of these BCPs can secure OAuth/OIDC implementations in the native apps.

3.3.1.1 Authorization Code Flow for Native Apps

The *Authorization Code* flow with the Proof Key for Code Exchange (*PKCE*) [SBA15], illustrated in Figure 3.2, encompasses the following steps:

1. *C* initiates the flow by directing the *RO*'s *UA* to the authorization endpoint including the *Authorization Request* query parameters, namely $OIDC_A$ and $PKCE_A$.
2. *AS* authenticates *RO* by displaying the login and consent form in the *UA*.
3. Assuming the access grant acceptance by the *RO*, *AS* redirects the *UA* back to *C* with an Authorization `code`.
4. *C* sends the Token Request ($OIDC_T$) together with *PKCE* query parameters ($PKCE_T$) to *AS* asking for the OIDC tokens.
5. *AS* executes the *PKCE* checks, If it is successful, it releases two OIDC tokens: `access_token` of type token_type (usually `Bearer`) with an expiration indicated in the field `expires_in`, and `id_token`, a signed JWT that contains basic attributes about *RO*.

Note that, Figure 3.2 represents the *Authorization Code* flow in OIDC that is used as a reference model to incrementally explicit the changes that app developers need to consider in case of iGov (Section 3.3.2) and FAPI (Section 3.3.3) profiles. As OIDC is built on top of OAuth, all the following details are applied for OAuth *Authorization Code* flow as well. The only minor change is related to the `scope` query parameter, which does not include the `oidc` value in the case of OAuth flow.

Table 3.1 summarizes the OIDC query parameters of *Authorization Code* flow with *PKCE* for native apps.

Table 3.1: OIDC query parameters of Authorization Code Flow with *PKCE* for Native Apps.

Tag	Query Parameter	Value	Description
$OIDC_A$	`response_type`	`code`	Determines the authorization flow to be used
	`scope`	`oidc`	A space-separated string of the scopes being requested and the `oidc` must be present when OIDC is supported
	`client_id`	string	The unique identifier of the native app
	`redirect_uri`	string	The URL to return to after the successful authentication
$PKCE_A$	`code_challenge`	string	A string generated by applying `code_challenge_method` to the `code_verifier` value
	`code_challenge_method`	`plain` or `S256`	Transformation method used to calculate `code_challenge`
$OIDC_T$	`grant_type`	`authorization_code`	Determines the way *C* gets the OIDC-Core tokens
	`code`	string	The authorization code received in Step 3 from *AS*
$PKCE_T$	`code_verifier`	string	A high-entropy cryptographic random string

3.3.1.2 Security Considerations

To extract the OAuth/OIDC security BCPs, we started from the RFC8252 entitled "OAuth 2.0 for Native Apps" [DB17], which has been published by the OAuth WG to assist app developers with secure integration of OAuth/OIDC solutions within their apps. In addition, the OAuth WG has recently published a new security BCPs document in [LBLF21]. The document updates and extends the previously defined threat model [LMH13] to incorporate practical experiences gathered since OAuth was published. Thus, we updated the extracted security BCPs from RFC8252 [DB17] with the latest suggestions for native apps in [LBLF21]. In the following, we explain each updated security BCPs that we initially extracted from the "OAuth 2.0 for Native Apps" specification [DB17].

- BCP_{flow}. The OAuth WG defines four grant types, namely: *RO Password Credentials, Implicit, Authorization Code* and *Client Credentials* (see Section 2.2.1). However, BCPs enforce the usage of the *Authorization Code* flow without `client_secret` (BCP_{flow}) among the other grant types because it can be protected by *PKCE* (BCP_{PKCE}), while it is not the case for other grant types. In addition, as all information passing through a mobile device is visible to *RO*, the confidentiality of the `client_secret` is not guaranteed (an app is a public client). Therefore, compared to the web scenario, there is no demand to send the `client_secret` alongside the authorization `code` to ask for the token.

- BCP_{UA}. In the Android environment, two kinds of *UA* are supported: embedded (e.g., Web View) and external (e.g., OS browsers app, in-app browser tabs (Custom Tabs), and IdMP apps) (Section 2.2.3). However, the BCPs strongly discouraged the usage of embedded *UA*s by third-party apps as it violates the principle of the least privilege by having access to the user's full authentication credential beside the OAuth authorization grant credentials (e.g., Authorization `code`). Therefore, BCPs recommend the usage of external browsers (BCP_{UA}). In this way, given the isolation they provide, the *C* app cannot access the cookie storage, neither inspect nor modify the page content.

- BCP_{redir}. One of the main peculiarities of the native scenario in comparison with the web scenario (discussed in Section 2.2.3) is the way that mobile platforms are receiving the redirection between *AS* and *C*. There are several redirection options available for apps in the Android environment: Custom URI scheme, app-claimed HTTPS (hereafter "HTTPS scheme" or simply "HTTPS" when it is clear from the context), and Loopback Interface. However, BCPs recommend the usage of either HTTPS or Custom URI (BCP_{redir}). Furthermore, BCPs recommend the usage of HTTPS over the Custom URI (if possible), as it provides a way to guarantee the identity of the destined app to *AS* and ensure the confidentiality of the Authorization `code` for the *C* app. While this is not the case for the Custom URI scheme, as multiple apps can typically register the same Custom URI scheme that makes it indeterminate for OS to which app it should redirect the Authorization `code`.

Thus, the OAuth WG introduces *PKCE* query parameters to address this limitation that is detailed in the BCP_{PKCE}.

- BCP_{PKCE}. The OAuth WG enforces the usage of *PKCE* as a proof-of-possession to protect the Authorization `code` from being used in case it is intercepted due to the usage of the Custom URI scheme (BCP_{PKCE}). Thus, the *C* app in the token request must send, together with the Authorization `code`, the `code_verifier` created at the beginning of the flow. Then, *AS* applies the code challenge method transformation `t_m` to `code_verifier` and compares the result with the code challenge value `t(code_verifier)` received in the authorization request. Only if the value is the same, then *AS* releases the (`access_token` and `id_token`) to *C*. Note that the OAuth WG also enforces the usage of BCP_{PKCE} in the case of HTTPS redirection because of the security benefits that it provides against code injection and code replay attacks.

Table 3.2 summarizes the key points concerning the security BCPs mentioned above for native apps. In the following, we explain how the enforcement of security BCPs can help mitigate the threats reported in Section 3.2.

Table 3.3 shows how the BCPs reported in Table 3.2 can be used to mitigate the considered threats. Concerning the notation used in Table 3.3, we point out as follows: (i) as a stolen Authorization `code` can be used by an attacker to obtain a valid `access_token`, the countermeasures related to obtaining the Authorization Code (indicated as $C(T_{code})$ in Table 3.3) indirectly mitigate the obtaining Access_tokens (T_{token}); (ii) the OR is inclusive, meaning that both the mitigations can be used at the same time to improve the protection; (iii) BCP_{redir}=HTTPS refers to the case in which the HTTPS scheme is used. In the following, we briefly explain the content of Table 3.3.

- As reported in Table 3.2, BCPs enforce the usage of the *Authorization Code* flow without `client_secret` (BCP_{flow}). The *Authorization Code* flow, unlike the *Implicit* flow, mitigates T_{token}, as it does not cause IdMPs issuing `access_token` in the authorization response. Furthermore, the `client_secret` is not used and T_{secret} is thus mitigated, given that apps cannot keep confidential the `client_secret`.

Table 3.2: OAuth/OIDC Security BCPs for Native Apps.

	Best Current Practice
BCP_{flow}	*Authorization Code* without `client_secret`
BCP_{UA}	External (possibly "in-app browser tabs")
BCP_{redir}	app-claimed HTTPS or Custom URI
BCP_{PKCE}	MUST use *PKCE* in their implementation

- BCPs recommend the usage of external browsers (BCP_{UA}), as reported in Table 3.2 to mitigate T_{code} and T_{cred}, because they provide an isolation, so the *C* app cannot access the cookie storage, neither inspect nor modify the page content.

- BCPs recommend the usage of either HTTPS or Custom URI (Table 3.2). As HTTPS redirection provides a way to guarantee the identity of destined app to *AS* and ensure the confidentiality of the Authorization `code` for the *C* app. Thus, BCP_{redir}=HTTPS mitigates T_{code} (and then T_{token}). While this is not the case for Custom URI scheme, as multiple apps can typically register the same Custom URI scheme that makes it indeterminate for OS to which app it should redirect the Authorization `code`. This limitation can lead to T_{token} if additional countermeasures are not implemented. Thus, Custom URI scheme must be used together with BCP_{PKCE} to mitigate T_{token}.

- As reported in Table 3.2, the OAuth WG enforces the implementation of *PKCE* (BCP_{PKCE}). In this way, it provides a proof-of-possession to protect the Authorization `code` from being used in case it is intercepted due to the usage of the Custom URI redirection. Thus, *PKCE* mitigates T_{token}. In addition, as the *PKCE* `code_verifier` query parameter is a unique per request random value, it provides linkability between the authorization request/response, and so it mitigates $T_{session}$. Note that to mitigate $T_{session}$, app developers must be sure that the IdMP supports *PKCE* before using it. Otherwise, the OAuth WG recommends the proper usage of the `state` and/or `nonce` query parameters [LBLF21]. `state` is an unique opaque value per request, which is sent by the client in the authorization request and the same value returns by *AS* during the redirection to *C*. Thus, similar to the *PKCE*, it provides a way to link the authorization request and response. This query parameter is a traditional way to avaoid $T_{session}$ as mentioned in [LMH13]. Furthermore, OIDC existing `nonce` query parameter can also be used for the same purpose as it is one-time use and created by the client [LBLF21].

As a final remark concerning the aforementioned countermeasures, it is worth mentioning that even if a *C* app does not implement all of them, this does not necessarily mean that the OAuth/OIDC implementations are vulnerable. Indeed, the aforementioned security BCPs provide

Table 3.3: Threats, OAuth/OIDC Security BCPs countermeasures, and Attacks.

Threats		Countermeasures	Attack examples
T_{token}	Obtaining Access Token	BCP_{flow} AND $\big(C(T_{code})$ OR $BCP_{PKCE}\big)$	Token Leakage
T_{code}	Obtaining Authorization `code`	BCP_{redir}=HTTPS AND BCP_{UA}	Code Eavesdropping
T_{cred}	Obtaining User Credential	BCP_{UA}	Webview Hijacking
T_{secret}	Obtaining Client Secret	BCP_{flow}	Client Impersonation
$T_{session}$	Session Misuse	BCP_{PKCE}	Login CSRF

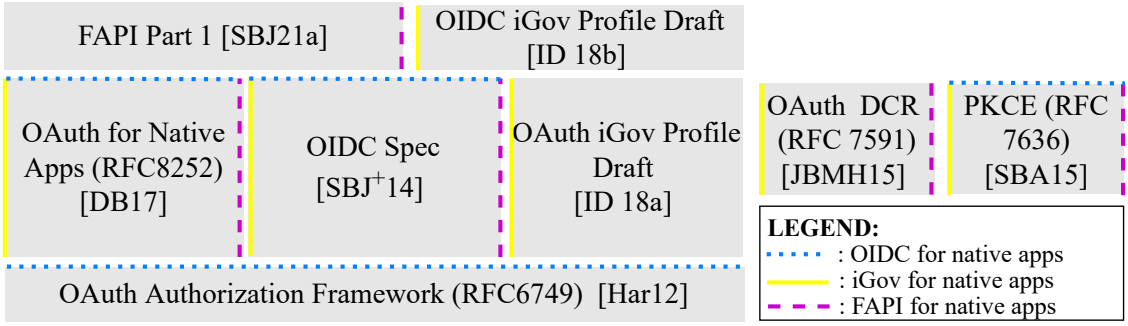


Figure 3.3: The Relation among OAuth, OIDC, iGov, and FAPI profiles.

sufficient security mechanisms to mitigate the above threats, but it is not strictly necessary to apply them: *AS* and *C* app developers may implement other appropriate mechanisms to implement OAuth/OIDC securely. However, we cannot conclude that the adoption of alternative strategies made their implementation secure.

3.3.2 iGov Security Best Current Practices

As mentioned in Section 2.4.1, the iGov profile is designed on top of the OAuth/OIDC core standards to provide a higher security level in the governmental domains. To extract the BCPs for iGov profile, we revisit the OAuth/OIDC iGov profiles [ID18a, ID18b] as an extension of OAuth for native apps.

In Section 3.3.2.1, we explain the iGov flow for native apps to summarize all the new query parameters that are used in iGov profile to provide a higher security level and Section 3.3.2.2 summarizes the main considerations w.r.t. iGov BCPs concerning the increased security achieved in iGov profile.

3.3.2.1 iGov Authorization Code Flow for Native Apps

Figure 3.3 represents a simplified relation among OAuth, OIDC, and iGov profile indicated with yellow line. Similarly to OIDC [SBJ+14], the iGov profile was designed without having in mind the native scenario. To illustrate the main concepts related to the iGov for native apps, we first explain the two client registration methods (namely, *static* and *dynamic*) supported by the iGov profile. Then, we describe the two iGov flows for the native apps by detailing the appropriate client authentication methods. Finally, we compare these two flows with the OIDC flow for native apps (Figure 3.2) to incrementally explicit the changes that app developers must consider (e.g., mandatory query parameters in authorization/token request).

The OAuth iGov profile [ID18a], which OIDC iGov profile [ID18b] is based on, specifies two

client registration methods:

Static Client Registration: app developers obtain a single `client_id` for a *C* app from the IdMP during the client registration phase at the IdMP developer console.

Dynamic Client Registration: app developers obtain a different `client_id` for each instance of the *C* app. IdMPs may implement the DCR endpoint as described in Section 2.3.2. App developers submit a new client registration request by setting the following values within the request to obtain per app instance `client_id`:

- `application_type` must be set to `native`;
- JSON Web Key Set (`jwks`) must include the *C*'s public keys;
- `token_endpoint_auth_method` must be set to `private_key_jwt`;
- `response_type` and `grant_type` can be ignored in the DCR request as their default values are equal to `code` and `authorization_code`, respectively; `redirect_uri` must be set into the URL to return to after the registration.

While the static client registration is the recommended method for public clients that do not have a back-end to securely store the secrets, the DCR is recommended wherever a back-end is existing. Based on the two aforementioned client registration methods, the iGov profile recommends different ways to authenticate apps at the token endpoint during the iGov flow (see Table 3.4); we provide more detail in the corresponding section below.

Considering the above-mentioned client registration methods, we explain the two iGov flows for native apps by detailing the client authentication method regarding their corresponding client registration methods.

iGov following Static Client Registration. As it is shown in Table 3.4, in case of static client registration, *C* must be considered as a public client, i.e., not capable of storing a client secret securely. Thus the only (mandatory) option is the usage of *PKCE.*

Compared to the flow of Figure 3.2, the iGov flow following the static client registration shows two differences in the authorization request (Step 1): (i) `nonce` is added as mandatory query

Table 3.4: iGov Scenario.

	Client Registration Method	**Client Authentication Method**	
		PKCE	**Private Key JWT**
iGov	Static CR	✓	×
	Dynamic CR	Opt	✓

parameter. Being present in the authorization request, this string will be returned in the `id_token` to mitigate replay attacks; and (*ii*) `code_challenge_method=S256`. This means that the `code challenge` value will be always calculated performing a SHA256 hash of the `code_verifier` string.

iGov following Dynamic Client Registration. In this scenario, as a consequence of DCR, *C* is considered an entity that can securely handle the client credentials (public/private key) created on the user's device. In this case, as shown in Table 3.4, the client authentication is based on the `private_key_jwt` method as preliminary specified in the `token_endpoint_auth_method` query parameter exchanged during the DCR (to obtain the `client_id`). The `private_key_jwt` method requires *C* to send a JWT signed with its private key when requesting access tokens. Thus, a token request will contain the following query parameters: (*i*) `client_assertion` is a single JWT that is signed by using the private key of the *C* app and its structure is detailed in Table 3.5; and (*ii*) `client_assertion_type` defines the type of assertion that is used by the *C* app.

Besides, DCR may use *PKCE* even if it is not mandatory.

The iGov flow following the DCR encompasses three differences compared to the flow of Figure 3.2:

- In the authorization request (Step 1), beside the addition of the `nonce` query parameter, the $PKCE_A$ query parameters may be removed.
- In the token request (Step 4), if the *C* app uses the *PKCE*, then the $PKCE_T$ query parameters are provided together with the query parameters required for the `private_key_jwt` client authentication method, which are `client_assertion` and `client_assertion_type`. Otherwise, in the token request, only the `private_key_jwt` client authentication query parameters are present.
- In Step 5, if the *PKCE* is not present, then IdMP only verifies the `client_assertion` value. Otherwise, IdMP must perform both *PKCE* and `client_assertion` checks.

Table 3.5: `client_assertion` JWT Structure.

Key	Value	Description
`iss`	String	`client_id` of the *C* app creating the request
`sub`	String	`client_id` of the *C* app creating the request
`aud`	String	the URL of the IdMP's token endpoint
`iat`	String	the issue time of the *C* request
`exp`	String	the *C* request expiration time
`jti`	String	a unique 128 bits identifier generated by the *C* per request

3.3.2.2 Security Considerations

As mentioned in the beginning of this section, the iGov profile is designed on top of the OAuth/OIDC core standards to provide a higher level of security in the governmental domains. Thus, in addition to the security BCPs specifically recommended for iGov, all the security BCPs defined in Table 3.2 are applied for the iGov as well.

The main iGov BCPs considerations concerning the increased security achieved in iGov can be summarized as follows.

1. the iGov BCPs mandates IdMPs to support DCR and as reported in Table 3.6, to enforce the `private_key_jwt` client authentication method rather than `client_secret_basic` for dynamically registered clients ($BCP^{iGov}_{ClientAuthn}$). They are because of the following security advantages: (i) DCR releases separate credentials for each C instance, thus it limits the compromise to just one C instance instead of all Cs that may share the same credentials; and (ii) asymmetric client authentication methods like `private_key_jwt` provide a higher level of protection as reported in [LBLF21]. In addition, as reported in Table 3.7, $BCP^{iGov}_{ClientAuthn}$ contributes to mitigates T_{token} besides the default countermeasures provided by OAuth BCPs.

2. as reported in Table 3.6, the iGov BCPs mandates IdMPs to accept authorization request with the `request`, `acr_values` and `nonce` query parameters, while they are optional in the OIDC core standard. The `request` query parameter ($BCP^{iGov}_{request}$) enables the OAuth/OIDC requests to pass in a single, self-contained query parameter and to be optionally signed and/or encrypted. Thus, it avoids message tampering within the browser and provides message integrity that helps to mitigate T_{code} as reported in Table 3.7. The `acr_values` query parameter (BCP^{iGov}_{acr}) enables IdMPs to request strong authentication methods to harden

Table 3.6: iGov Security BCPs for Native Apps.

Tag		Best Current Practice
$BCPs_{OAuth}$	BCP_{flow}	*Authorization Code* without `client_secret`
	BCP_{UA}	External (possibly "in-app browser tabs")
	BCP_{redir}	app-claimed HTTPS or Custom URI
	BCP_{PKCE}	Must use *PKCE* in their implementation
$BCPs_{iGov}$	BCP^{iGov}_{Nonce}	MUST use `Nonce` in their implementation
	$BCP^{iGov}_{ClientAuthn}$	MUST use `Private_Key_JWT` for dynamically registered client
	$BCP^{iGov}_{request}$	MAY use `request` query parameter in their implementation
	BCP^{iGov}_{acr}	MAY use `acr_values` query parameter in their implementation

intrusion attempts against users by mandating additional authentication factors. Thus, `acr_values` mitigates T_{cred}. Finally, `nonce` query parameter (BCP^{iGov}_{Nonce}) provides additional countermeasures to mitigate $T_{session}$ and T_{token} as mentioned in Table 3.7. Note that, as reported in Table 3.6, the iGov profile enforces the implementation BCP^{iGov}_{Nonce} (reported with "MUST"), while the implementation of $BCP^{iGov}_{request}$ and BCP^{iGov}_{acr} are recommended (reported with "MAY").

3.3.3 FAPI Security Best Current Practices

The FAPI profile—indicated with purple dashed line in Figure 3.3— is designed on top of OAuth and OIDC core standards to provide a higher level of security rather than the baseline security provided by the OAuth core or OIDC core [SBJ21a, SBJ21b]. As mentioned in Background (Section 2.4.2), the FAPI WG introduces two security profiles, namely: Read-Only API Security Profile (FAPI-Part1), and Read and Write API Security Profile (FAPI-Part2). As FAPI-Part2 does not provide support for native apps (Section 5.1 in [SBJ21b]), we only considered the FAPI-Part1 [SBJ21a] document to extract security BCPs for FAPI profile.

Section 3.3.3.1 details the FAPI flow for native apps to summarize all the mandatory query parameters that are used in the FAPI profile to provide a higher level of security, and Section 3.3.3.2 explains the main considerations w.r.t. FAPI security BCPs concerning the increased security achieved by FAPI profile.

3.3.3.1 FAPI Authorization Code Flow for Native Apps

The FAPI profile [SBJ21a] similar to the iGov profile specifies two possible flows for native apps w.r.t. their client registration (namely, *static* and *dynamic*) and client authentication methods. Thus, all the modifications in authorization and token requests query parameters introduced by iGov (Section 3.3.2.1) are applicable in the case of FAPI with a slightly few modifications.

Table 3.7: Threats, iGov Security BCPs countermeasures, and Attacks.

Threats	Countermeasures	Attack examples
T_{token}	BCP_{flow} AND $\big(C(T_{code})$ OR $BCP_{PKCE}\big)$ AND $BCP^{iGov}_{ClientAuthn}$ AND BCP^{iGov}_{Nonce}	Token Leakage
T_{code}	BCP_{redir}=HTTPS AND BCP_{UA} AND $BCP^{iGov}_{request}$	Code Eavesdropping
T_{cred}	BCP_{UA} AND BCP^{iGov}_{acr}	Webview Hijacking
T_{secret}	BCP_{flow}	Client Impersonation
$T_{session}$	BCP_{PKCE} AND BCP^{iGov}_{Nonce}	Login CSRF

In particular, the FAPI profile introduces the following additional considerations that app developers must consider within their implementation:

Static Client Registration: FAPI profile allows only "HTTPS" redirect URI registration during client registration on the IdMP developer console to obtain a single `client_id`, while iGov profile allows both HTTPS and Custom URI scheme;

Dynamic Client Registration: the main difference in the DCR request for FAPI profile in comparison with iGov is as follows:

- FAPI profile allows `client_secret_jwt` and `Mutual TLS` (`mTLS`) besides `private_key_jwt` introduced by iGov (Section 3.3.2.1) to be set as a value for the `token_endpoint_auth_method`; Note that, `mTLS` defines two method for using client certificates for client authentication, namely PKI `mTLS` and Self-Signed `mTLS` [CBSL19];
- if `mTLS` selected as a client authentication method, app developers need to register the client certificate through the DCR request. Interested readers can obtain more information w.r.t. registering the client certificate in [CBSL19].

The above-mentioned modifications for client registration lead to the following changes in the FAPI flow for native apps:

FAPI following Static Client Registration: compared to the modifications introduced by the iGov following Static Client Registration (Section 3.3.2.1), FAPI profile additionally enforces app developers to only use HTTPS for redirection;

FAPI following Dynamic Client Registration: The main difference in FAPI following Dynamic Client Registration with the iGov flow, introduced in Section 3.3.2.1, for the dynamically registered client is the possibility to use `secret_key_jwt` and `mTLS` in addition to `private_key_jwt` for the client authentication at token endpoint. While `secret_key_jwt` requires *C* to pass a JWT signed with `client_secret` by using a symmetric algorithm when requesting access tokens [SBJ+14], in case of `mTLS` [CBSL19] *C* requires to send the client certificate used in a TLS connection between the client and the token endpoint for client authentication.

3.3.3.2 Security Considerations

As represented in Figure 3.3, the FAPI profile is designed on top of OAuth and OIDC core standards. Thus, all the security BCPs defined in Table 3.2 are applied for FAPI except the one related to the BCP_{redir}. Table 3.8 reports the main FAPI security BCPs considerations concerning the increased security achieved in FAPI. They can be summarized as follows :

1. the FAPI profile mandates HTTPS as an only option for the redirection (BCP^{FAPI}_{redir}), as it provides a way to ensure the confidentiality of Authorization `code` for the *C* app. Thus, it can help to mitigate T_{code} as reported in Table 3.9.

2. the FAPI profile enforces BCP^{FAPI}_{Nonce} in case of OIDC implementation and `state` query parameter in the case of OAuth implementation to mitigate $T_{session}$.

3. the FAPI profile mandates IdMPs to support DCR, and recommends one of the following [`mTLS`, `client_secret_jwt` and `private_key_jwt`] client authentication methods in the case of dynamically registered clients ($BCP^{FAPI}_{ClientAuthn}$). The main reason is that these methods provide a higher security level in comparison with `client_secret_basic` and they can help in mitigation of T_{token} as reported in Table 3.9.

Summary. To summarize the concepts introduced in the previous sections, we provide a comparison among the OAuth/OIDC core, iGov and FAPI profiles BCPs from a security perspective. Below, we provide a brief explanation concerning the main differences reported in Table 3.10:

- OAuth/OIDC core, iGov and FAPI profile security BCPs all recommend the adoption of

Table 3.8: FAPI Security BCPs for Native Apps.

Tag		Best Current Practice
$\text{BCPs}_{\text{OAuth}}$	BCP_{flow}	*Authorization Code* without `client_secret`
	BCP_{UA}	External (possibly "in-app browser tabs")
	BCP_{PKCE}	Must use *PKCE* in their implementation
$\text{BCPs}_{\text{FAPI}}$	BCP^{FAPI}_{redir}	app-claimed HTTPS
	BCP^{FAPI}_{Nonce}	Must use `Nonce` in their implementation
	$BCP^{FAPI}_{ClientAuthn}$	Must use `private_Key_JWT` or `client_secret_jwt` or `mTLS` for dynamically registered client

Table 3.9: Threats, FAPI Security BCPs countermeasures, and Attacks.

Threats	Countermeasures	Attack examples
T_{token}	BCP_{flow} AND $\big(C(T_{code})$ OR $BCP_{PKCE}\big)$ AND $BCP^{FAPI}_{ClientAuthn}$ AND BCP^{FAPI}_{Nonce}	Token Leakage
T_{code}	BCP^{FAPI}_{redir} AND BCP_{UA}	Code Eavesdropping
T_{cred}	BCP_{UA}	Webview Hijacking
T_{secret}	BCP_{flow}	Client Impersonation
$T_{session}$	BCP_{PKCE} AND BCP^{FAPI}_{Nonce}	Login CSRF

BCP_{flow}, BCP_{UA}, and BCP_{PKCE};

- OAuth/OIDC core and iGov profile BCPs recommend adoption of either Custom URI or HTTPs redirection methods for the BCP_{redir} (represented by vertical "MUST"), while the FAPI profile BCPs only recommends the usage of HTTPs redirection method;
- BCP_{acr} is not applicable within OAuth/OIDC core, while it is optional for iGov and FAPI profiles BCPs;
- $BCP_{request}$ is not applicable within OAuth/OIDC core and FAPI profile BCPs, while it is optional for iGov profile BCPs;
- iGov and FAPI profiles BCPs mandate the adoption of BCP_{Nonce}, while it is not applicable in the OAuth/OIDC core BCPs;
- The FAPI and iGov profile BCPs recommend the adoption of $BCP_{ClientAuthn}$, while this is not the case for OAuth/OIDC core BCPs. FAPI profile BCPs recommends the implementation of one of the following [`mTLS`, `client_secret_jwt` and `private_key_jwt`] client authentication methods (represented by vertical "MUST"), while iGov profile BCPs recommends the usage of `private_key_jwt` among the aforementioned client authentication methods.

3.3.4 Our Recommended Privacy Best Current Practices

While OAuth core documentation does not provide any specific privacy consideration section, OIDC core, FAPI profile, and iGov profile provide a section specifically related to privacy consid-

Table 3.10: OAuth/OIDC core, iGov, FAPI profiles Security BCPs Comparison.

	OAuth BCPs	iGov BCPs	FAPI BCPs
BCP_{flow}	MUST	MUST	MUST
BCP_{UA}	MUST	MUST	MUST
BCP_{PKCE}	MUST	MUST	MUST
BCP_{redir}=`HTTPS` **OR**	MUST	MUST	MUST
BCP_{redir}=`Custom URI`			N/A
$BCP_{ClientAuthn}$=`private_key_jwt` **OR**	N/A	MUST	MUST
$BCP_{ClientAuthn}$=`client_secret_jwt` **OR**	N/A	N/A	
$BCP_{ClientAuthn}$=`mTLS`	N/A	N/A	
BCP_{Nonce}	N/A	MUST	MUST
BCP_{acr}	N/A	OPTIONAL	OPTIONAL
$BCP_{request}$	N/A	OPTIONAL	N/A

erations concerning OAuth/OIDC implementations. However, the provided information is either very abstract or does not provide specified recommendations on how to increase the privacy of users. Thus, we provide easy-to-implement privacy recommendations to assist app developers with providing OAuth/OIDC solutions that address the privacy requirements of users. It is worth mentioning that the following recommendations apply to both native and web scenarios.

Section 3.3.4.1details the privacy goals that app developers must satisfy within their OAuth/OIDC solutions and Section 3.3.4.2 explains how we use OAuth/OIDC optional query parameters and the ones introduced in OIDC iGov Profile [ID18b], OIDC for Identity Assurance [LF20], and Vector of Trust [RJ18] documents to address these privacy goals.

3.3.4.1 Privacy Goals

Security goals (*Confidentiality*, *Integrity*, *Availability*) need to be complemented with further privacy goals [HJR15] to evaluate the impact on all aspects of user privacy. They are: *Data unlinkability*, *Data minimization*, *Purpose specification*, *Transparency*, and *Intervenability*. Recent research efforts have come up with these goals [RP09, DDFH$^+$15, RB11, HJR15] to provide an interdisciplinary standard model to assess the consequences of a complex IT systems concerning privacy [HJR15]. Below, we provide a brief explanation for the CIA and the above mentioned privacy goals:

- *Confidentiality* (C) safeguards authorized restrictions on information access and disclosure and it provides a way to protect personal privacy and proprietary information [NDP$^+$17];
- *Integrity* (I) protects data and/or system from improper information modification and preserves information non-repudiation and authenticity [NDP$^+$17];
- *Availability* (A) ensures timely and reliable access to and use of information [NDP$^+$17];
- *Data unlinkability* refers to hiding the link between two or more actions, identities, and pieces of information [WSJ$^+$19];
- *Data minimization* requires avoiding unnecessary data to achieve the determined purpose, that is, *purpose specification* [DDFH$^+$15];
- *Transparency* requires data processing to be understandable and reconstructable by concerned individual [DDFH$^+$15];
- *Intervenability* requires that intervention (for the individual whose data are processed) is possible concerning all ongoing or planned privacy-relevant data processing [DDFH$^+$15].

3.3.4.2 Privacy Considerations

In the following, we explain how we used the optional OAuth/OIDC query parameters and the ones introduced in OIDC iGov Profile [ID18b], OIDC for Identity Assurance [LF20], and Vector of Trust [RJ18] documents to address the reported privacy goals.

Confidentiality and Data Accuracy. `Acr_values` and *Vector of Trust* (`vot`) [SBJ+14, RJ18] query parameters provide a way to request IdMPs to guarantee a certain level of identity proofing. While OIDC `acr_values` is a singular scalar value, `vot` provides more actionable data and expressiveness to convey trust information by defining four orthogonal components. Below, we briefly define the meaning of each `vot` components, and interested readers can refer to [RJ18] for additional information:

- Identity Proofing (P): to assess how strongly the IdMP verified attributes that users provided to get authenticated. This component has values ranging from `P0` to `P3`. For example `P1` means that users attributes are self-asserted, while *P2* means that the user attributes have been proofed either in person or remotely using a trusted mechanism (e.g., social proofing). Interested readers can refer to [RJ18] for further information. It is worth mentioning that as higher level values provide a stronger method for user attributes assessment, thus they satisfy the lower level data accuracy requirements, e.g., *P2* fulfills the requirements for *P1*. Note that only one distinct value from this category must be used within OAuth/OIDC solutions;

- primary credential assurance (C): to assess how strongly IdMP has verified the means users used to authenticate. In simple words, it represents how easily the user credentials could be spoofed or stolen. This component defines the following values: [`C0, Ca, Cb, Cc, Cd, Ce, Cf, Cg`]. To illustrate, we provide the definition for the following `Cc`, and `Cd` values and interested readers can obtain definitions of other values in [RJ18]. For example `Cc` represents the usage of shared secret (e.g., username and password combination) , while `Cd` means cryptographic proof of key possession using shared key. Note that multiple values from this category may be used within OAuth/OIDC solutions;

- primary credential management (M): to convey information about the expected life cycle of means users used to authenticate. In particular, it verifies the strength of policies, practices, and security controls used in managing the credentials at the IdMP and its binding to the intended users. This component has the following values [`Ma, Mb, Mc`]. To illustrate, let us define `Ma` value and the other values definition are available for interested readers in [RJ18]. For example `Ma` means that the user chooses its own credentials and must rotate and revoke them manually. In addition, there is no need for further verification for primary credential issuance or rotation. It is worth mentioning that multiple values from this category may be used in OAuth/OIDC solutions; and

- assertion presentation (A): to assess how well the digital identity can be communicated across the network without information leaking to unintended parties and without spoofing. This component has the following values [Aa, Ab, Ac, Ad]. In the following, I explain the Ab value and interested readers can refer to [RJ18] document to find definitions of remaining values. For example Ab means that the assertions provided by IdMP is signed and verifiable. In addition, they passed through the *UA*.

In the following, we provide our recommended values for vot query parameter concerning the privacy requirements of developers in their OAuth/OIDC solutions. Needless to say, as mentioned in Table 3.11, app developers can use higher values rather than the ones we recommend based on their client operating domain. We suggest using [P2.Ac.Cd.Mb] for addressing the minimum level , and [P3.Ac.Ce.Mc] to satisfy a higher level for *Confidentiality and Data Accuracy* privacy goals. [P2.Ac.Cd.Mb] requires:

- to prove the identity either in person or remotely using a trusted mechanism, as it provides the minimum requirement for stringent proofing that the digital identity corresponds to a real user (P2).
- the signed and verifiable assertion needs to be passed through a back channel, as it is more secure rather than passing through the *UA*. Thus, it is less probable to leak assertions to unintended parties or get spoofed (Ac).
- cryptographic proof of key possession using the shared key for primary credential usage, as it is harder to get spoofed by the adversaries (Cd).
- remote user credentials issuance and rotation/use of backup recovery (e.g., email verification) by using value, as it enforces basic policies and security controls in managing the credential at the IdMP and its binding to the intended user (Mb) .

[P3.Ac.Ce.Mc] requires:

- a binding relationship between the IdMP and the identified party (P3).
- the same value is used for assertion presentation as described before (Ac).
- cryptographic proof of key possession using an asymmetric key due to the security benefits of asymmetric cryptographic methods compared to symmetric ones (Ce).
- proofing for user credentials issuance and rotation/revocation on suspicious activities to enforce stronger policies and security controls in managing the credential at the IdMP and its binding to the intended user Mc.

Transparency. In order to satisfy the *Transparency* privacy goal within the OAuth/OIDC implementations, we recommend the usage of OIDC *prompt* and Identity Assurance *purpose* query parameter in [SBJ+14, LF20]. The *prompt* query parameter provides a way for users to re-authenticate and collect their consent before returning information from IdMP by setting this query parameter to `consent`. While the *purpose* query parameter provides a way to display the purpose of requesting indivual claims by developers in the respective Identity Provider's consent screen that fulfills also privacy goal *Transparency*.

Accuracy, and Purpose Specification. The Identity Assurance [LF20] defines a generic mechanism through defining a new element called `verified_claim` to provide the app developer with a set of claims about users (e.g., name, and address) and the respective metadata and verification evidence to verify them. This way, app developers cannot mix up verified and unverified claims. The `verified_claims` is an object or array containing one or more verified claims objects. A single `verified_claims` object consist of the following sub-elements:

verification. represents an object that provides information about the verification process. The `verification` element can consist of optional and mandatory elements, such as `trust_framework` (mandatory), `time` (optional), and `evidence` (optional).

- `trust_framework` defines the identity verification process and the identity assurance level of the Identity Provider (e.g., eidas_ial_high).
- `time` element represent the date and time when the identity verification process took place.
- `evidence` is an JSON array that provides information concerning the evidence used by Identity Provider to verify the user's identity. it must contain the property `type` that determines the type of the evidence, and may contains additional sub-property based on the evidence type. The following types of evidence are defined, which are: id document, utility bill, and eIDAS Qualified Electronic Signature (qes). Interested readers can refer to the Identity Assurance [LF20] document for additional information concerning `verification` element.

Table 3.11: Privacy BCPs.

Privacy BCPs	Goal
`claims`	personal data minimization
`scope`	personal data minimization
`purpose`	purpose specification
`verified_claims`	personal data accuracy, integrity
`vot`:*P3.Ac.Cc.Mc/Higher*	personal data accuracy, personal data confidentiality
`acr_values`:*acr_level2/3*	personal data accuracy, personal data confidentiality
`consent`	transparency
`pairwise`	unlinkability

claims. represents the claims about the user which are verified in accordance with the policies determined by the corresponding `verification` element. The `claims` element can contains the standard users claims as defined in the OIDC core [SBJ+14] document (e.g., name) and the new users claims that are defined in the *Identity Assurance* [LF20] document (e.g.,place_of_birth). It is worth mentioning that the Identity Assurance document introduces an additional query parameter that is called `purpose` to allow app developers state the purpose for obtaining certain user claim from the Identity Provider.

As such, the aforementioned elements enable the app developer to achieve the following privacy goals: (i) *Accuracy*, by using `evidence` to ask for the trust framework governing the identity verification, the identity assurance level of the Identity Provider, and the evidence used to verify the user's identity; and using `time` to ask for the date and time when the Identity Provider verified the identity. (ii) *Purpose specification*, by using `purpose` to enable app developers to state the purpose of asking each individual claim. The app developer requests this claim like any other claim via the `claims` query parameter or as part of a default claim set identified by a `scope` value [LF20].

Data Minimization. To satisfy *Data minimization* privacy goal, app developers can use the OIDC `claims` query parameter [SBJ+14] to ask for specific user claims. Note that `acr_values` also limits the data based on the requested level of assurance.

Data Unlinkability. OIDC core [SBJ+14] introduces two types of subject identifier, namely: `public` and `pairwise`. The former assigns an identifier to an user within different app developers, while the latter provides one unique users identifier per app developer. To fulfill this goal, the app developer needs to use `pairwise` identifiers as suggested by the iGov profile [ID18b].

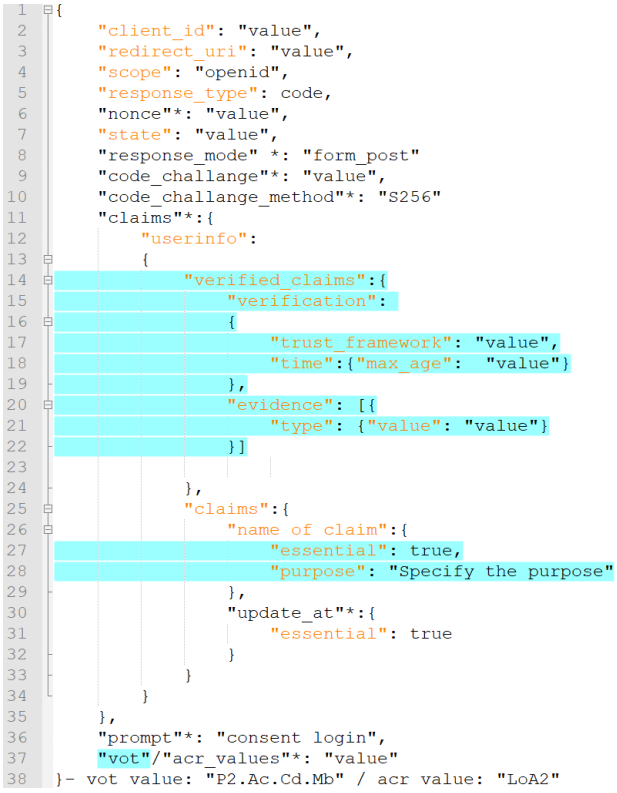
Unaddressed Privacy Goals. OIDC core states that "the users will be given the option to have IdMP decline to provide some or all information requested by app developers [SBJ+14]". that is privacy goal *Interveinability*. Meeting this goal depends on IdMP's implementation, and the app developer has no control over it. In addition, we do not consider *Accountability* and *Storage limitation* privacy goals as the OAuth/OIDC query parameters cannot provide any countermeasures for them.

OAuth/OIDC Compliance Request. We have generated two *authorization* and *token requests* that are compliant with previously mentioned BCPs for security and privacy. We consider OAuth security BCPs to satisfy the app requirements with the minimum level of security, and iGov/FAPI security BCPs to satisfy the app requirements with the higher level of security. Both schemas are compliant with privacy BCPs as the privacy goals shall be met either way.

Figure 3.4 represents BCPs-compliant authorization and token request. The query parameter indicated with an asterisk are the optional query parameters and the ones highlighted in blue are query parameters that are taken from the IdentityAssurance [LF20] and Vector of Trust [RJ18] documents.

```
{
    "client_id": "value",
    "request"*: jwt value
}
{// JWT Value:
    "redirect_uri": "value",
    "scope": "openid",
    "response_type": "code",
    "nonce"*: "value",
    "state": "value",
    "response_mode" *: form_post
    "code_challange"*: "value",
    "code_challange_method"*: S256
    "claims"*:{
        "userinfo":
        {
            "verified_claims":{
                "verification":
                {
                    "trust_framework": "value",
                    "time":{"max_age":  "value"}
                },
                "evidence": [{
                    "type": {"value": "value"}
                }]
            },
            "claims":{
                "name of claim":{
                    "essential": true,
                    "purpose": "Specify the purpose"
                },
                "update_at"*:{
                    "essential": true
                }
            }
        }
    },
    "prompt"*: "consent login",
    "vot"/ "acr_values"*: "value"
}- vot value: "P3.Ac.Ce.Mc" / acr value: "LoA2" or higher
```

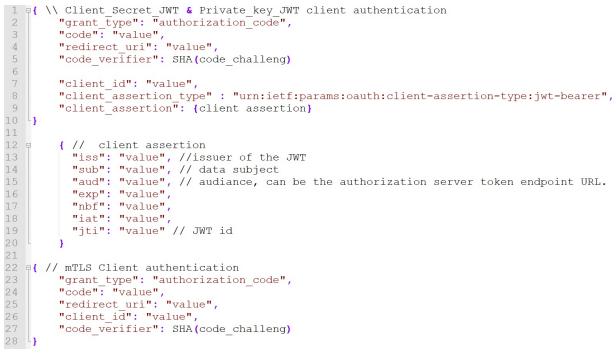
(a) Authorization Request: Sensitive Sector



```
{
    "client_id": "value",
    "redirect_uri": "value",
    "scope": "openid",
    "response_type": code,
    "nonce"*: "value",
    "state": "value",
    "response_mode" *: "form_post"
    "code_challange"*: "value",
    "code_challange_method"*: "S256"
    "claims"*:{
        "userinfo":
        {
            "verified_claims":{
                "verification":
                {
                    "trust_framework": "value",
                    "time":{"max_age":  "value"}
                },
                "evidence": [{
                    "type": {"value": "value"}
                }]

            },
            "claims":{
                "name of claim":{
                    "essential": true,
                    "purpose": "Specify the purpose"
                },
                "update_at"*:{
                    "essential": true
                }
            }
        }
    },
    "prompt"*: "consent login",
    "vot"/"acr_values"*: "value"
}- vot value: "P2.Ac.Cd.Mb" / acr value: "LoA2"
```

(b) Authorization Request: Non-Sensitive Sector



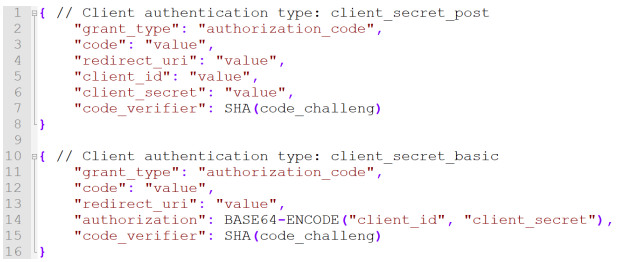
```
{ \\ Client_Secret_JWT & Private_key_JWT client authentication
    "grant_type": "authorization_code",
    "code": "value",
    "redirect_uri": "value",
    "code_verifier": SHA(code_challeng)

    "client_id": "value",
    "client_assertion_type" : "urn:ietf:params:oauth:client-assertion-type:jwt-bearer",
    "client_assertion": {client assertion}
}

    { //  client assertion
      "iss": "value", //issuer of the JWT
      "sub": "value", // data subject
      "aud": "value", // audiance, can be the authorization server token endpoint URL.
      "exp": "value",
      "nbf": "value",
      "iat": "value",
      "jti": "value" // JWT id
    }

{ // mTLS Client authentication
    "grant_type": "authorization_code",
    "code": "value",
    "redirect_uri": "value",
    "client_id": "value",
    "code_verifier": SHA(code_challeng)
}
```

(c) Token Request: Sensitive Sector



```
{ // Client authentication type: client_secret_post
    "grant_type": "authorization_code",
    "code": "value",
    "redirect_uri": "value",
    "client_id": "value",
    "client_secret": "value",
    "code_verifier": SHA(code_challeng)
}

{ // Client authentication type: client_secret_basic
    "grant_type": "authorization_code",
    "code": "value",
    "redirect_uri": "value",
    "authorization": BASE64-ENCODE("client_id", "client_secret"),
    "code_verifier": SHA(code_challeng)
}
```

(d) Token Request: Non-Sensitive Sector

Figure 3.4: BCPs-Compliant Request Schema.

Chapter 4

Compliance Analysis of IdMPs and Google Play Store Apps with OAuth and OIDC Security BCPs

This chapter presents our compliance analysis results for both popular IdMPs and top-ranked Google Play Store apps against the OAuth/OIDC security BCPs.[1]

4.1 Context and Motivations

As mentioned in Section 3.3, the OAuth WG and OIDF have published a set of security BCPs for native apps to assist native app developers with the secure integration of the OAuth/OIDC solutions. Thus, we extracted a set of security BCPs for native scenarios from the relevant OAuth/OIDC documents [DB17, LBLF21] used to secure native app developers implementation against a wide range of security threats. However, it is possible that native app developers are not aware of these security BCPs, as these documents have been published gradually over the years by the OAuth WG.

Given that, to verify the applicability and the role of aforementioned security BCPs (See Section 3.3.1) to increase the security of IdMP solutions and consequently their effect on the secure integration of IdMP's solution within native app developers app, we perform a survey on popular IdMPs and top-ranked Google Play Store Apps to check their current situation w.r.t. their compliance with OAuth/OIDC security BCPs for native apps.

Section 4.2 presents our selection procedure, identified and extracted features, and our compli-

[1]This chapter is based on our under revision work in [SCSR21].

ance analysis results for popular IdMPs, while in Section 4.3, we provide the current status in terms of compliance with OAuth/OIDC security BCPs as reported in Table 3.2 for native apps on the client (*C*) side by detailing the procedure for the app selection criteria, selected features and static analysis tools, and the analysis results.

4.2 Analysis of Popular OAuth/OIDC IdMPs

In this section, we present the current status in terms of compliance with OAuth/OIDC security BCPs for native apps of the most popular IdMPs. We performed a preliminary analysis in October 2019, and the analysis results are available in Appendix D. In the following, we report the outcome for the latest compliance analysis (February 2021) to check if there is an improvement in IdMPs solution compliance.

In Section 4.2.1, we briefly explain the selection criteria we followed to choose popular OAuth/OIDC IdMPs. In Section 4.2.2, we specify the features that we collected for each IdMP and briefly explain the steps that we followed to extract the features for the selected OAuth/OIDC IdMPs. Furthermore, we provide some insights concerning automating of IdMPs analysis. In Section 4.2.3, to analyze the selected features, we identify the relevant research questions, and for each research question, we report the results of our analysis.

4.2.1 IdMPs Selection

In our study, to select the popular IdMPs, we start from the lists of OAuth/OIDC IdMPs that are available on the OAuth [OAu19] and OIDC [Ope19] website.[2] Among the lists, we selected the top 10 IdMPs based on their Alexa rank and if they support the mobile native scenario. Then, we further harassed the list to keep the only one that the developer console is accessible without subscription. As a result, we have considered 14 IdMPs.

4.2.2 Feature Definition

To analyze the compliance with OAuth/OIDC security BCPs of IdMPs, for each selected IdMP (where for Microsoft we have analyzed two different versions[3]) we have extracted the following features (shown in Table 4.1):

[2]The OAuth List is provided by an OAuth WG community member and is not available anymore, it could be consulted by searching the site archive in Dec 2018.

[3]Microsoft provides two IdM APIs, namely: API v1 (ADAL), and API v2 (MSAL) for native app developers to integrate Microsoft SSO/AD solutions within their apps. Microsoft recommends native app developers to move into the MSAL, as the older version will no longer receive new feature improvements.

Scenario: describes if the IdMP supports *SSO* Login and/or *AD* scenarios;

OAuth/OIDC security BCPs compliance: specifies which *Flow* and *UA* types are supported by the SDK (if any) and if the *PKCE* solution is implemented.

To extract the features from the selected IdMPs, we manually analyzed their solution by checking either one of the following: IdMP's documentation, IdMP's developer console, and the IdMP's SDK (if available). The information related to the supported *Scenario* by each IdMP is obtained from the IdMP's documentation. To identify the supported *Flow* by the selected IdMP's SDK (if available), we look into their SDK for native apps. In the case of *PKCE*, we checked both the IdMP's SDK and developer console, as some of the selected IdMP's (e.g., IBM) allow developers to disable the *PKCE* support from the console. Finally, for the *UA*, we directly check the code within the IdMP's native SDK. The references to the online documentation and, if available, the SDK is reported after the IdMP name in Table 4.1.

Discussion. The above-mentioned manual analysis can be semi-automated to extract some of the features mentioned above using either static or dynamic analysis tools. While it is possible to some extent automate the detection of the *PKCE* and *UA* features, this is not the case for the *Flow*[4]. In the case of static analysis, good candidates are represented by SUPER and StaCon tools (See Appendix B). The former is suitable to detect the *UA* feature, the latter one is useful to detect *PKCE* by applying string matching techniques. Note that, we cannot detect whether *PKCE* support is optional.

Another solution for the above-mentioned problem is using of dynamic analysis techniques, which are capable of intercepting the traffic between native apps and IdMPs. A good example of such kind of tool is Micro-Id-Gym [BCPR20]. Micro-Id-Gym has been initially designed to check the implementation of identity management solutions within the web scenario. It is possible to extend the tool covering our demands for the native apps. We can adopt a string matching technique to extract the values for *PKCE* and *UA* features. For the former, we can look for the "`code_challenge`" field in the authorization request, and for the latter, we can check the user agent header. In particular, it contains "wv" in the user-agent header in Web View usage.

4.2.3 Research Questions and Results

To analyze the selected features, we identified the following research questions:

R1. How many IdMPs are violating OAuth/OIDC security BCPs?

R2. How many IdMPs are compatible with the AppAuth SDK?

[4]It is possible to perform the string matching to look for the relevant `response_type` strings, but we are not able to detect the proprietary implementations.

Table 4.1: IdMPs Status.

IdMP Name	Scenario		SDK Support		
	SSO	AD	Flows	UA	PKCE
Google [Goo21]	●	●	code w/o secret	CT	✓
Facebook [Fac21a, Fac21b]	◗	●	*modified implicit**	CT, *N**	N/A
Yahoo [Yah21]	●	●	-	-	-
Amazon [Ama21a, Ama21b]	◗	●	code w/o secret, *implicit**	Ex	✓, N/A
Linkedin [Lin21a, Lin21b]	◗	●	code *w/ secret*	Ex	x
Microsoft v1 [Mic21a, Mic21b]	●	●	code w/o secret	*Em*	*x**
Microsoft v2 [Mic21d, Mic21c]	●	●	code w/o secret	CT	✓
DropBox [Dro21a, Dro21b]	❍	●	code *w/ secret*	Ex	✓
OKTA [OKT21a, OKT21b]	●	◗	code w/o secret	CT	✓
Box [Box21a, Box21b]	❍	●	code *w/ secret*	*Em*, *N**	x, *x**
Auth0 [Aut21a, Aut21b]	●	◗	code w/o secret, *implicit*	CT	✓, N/A
HiDrive [HiD21a, HiD21b]	❍	●	*modified code w/ secret*	Ex	*x**
Beeminder [Bee21]	❍	●	-	-	-
Ping [Pin21a, Pin21b]	●	◗	code w/o secret, *implicit*	CT	✓‡
IBM [IBM21a, IBM21b]	●	❍	code w/o secret, *pass*	*N**	✓‡

Scenario	●	IdMP supports OIDC or OAuth
	❍	IdMP does not support OIDC or OAuth
	◗	IdMP supports a SSO Login different from OIDC or supports OAuth to access native app developer resources but does not expose its APIs
Flows	code	*Authorization Code* flow with (w/) or without (w/o) the use of `client_secret`
	implicit	*Implicit* flow
	pass	*RO Password Credential* flow
UA	Ex	External browser
	Em	Embedded browser
	CT	External browser supporting Custom Tab
	N	Native app
PKCE	✓	The SDK supports the *PKCE* protocol
	✓‡	The SDK supports *PKCE* but it is optional
	x	The SDK does not support *PKCE*
	x*	The SDK does not support *PKCE*, but it provides other security measures
	N/A	*PKCE* cannot be used as the flow is not the *Authorization Code* flow

In the following, we discuss each research question.

4.2.3.1 Discussion on *R1*

We use the collected data to analyze the selected IdMPs to check if they are compliant with OAuth/OIDC security BCPs for native apps (see Table 3.2). In Table 4.1, we use italic font to

indicate the features that are not compliant with OAuth/OIDC security BCPs. We mark with a red font the features causing security issues, while we use an asterisk to indicate that the IdMP provides other security measures to secure the flow.

Regarding the flow (4th column in Table 4.1), even if the *Authorization Code* flow with *PKCE* is the recommended flow (BCP_{flow}), IBM is also supporting the *RO Password Credential* flow and 4 IdMPs (Auth0, Ping, Beeminder, and Amazon) are still permitting the usage of *Implicit* flow, which is not recommended to use within apps as it cannot be protected by the *PKCE*. Note that, while the *Authorization Code* flow is the default flow for both Ping and Auth0 SDKs, there is still the possibility for an app developer to choose the *Implicit* flow. In addition, the Ping SDK API reference [Pin21a] suggests the usage of the *Implicit* flow as the default grant type because it enables inexperienced app developers to effortlessly learn and integrate OAuth/OIDC solutions within their apps. Therefore, the Ping suggestion to use *Implicit* flow for native apps may mislead app developers who intend to integrate the OAuth/OIDC solutions manually. Indeed, app developers may unintentionally integrate *Implicit* flow within their native apps, even if this is strongly discouraged by BCPs [DB17]. In the case of Amazon, the *Implicit* flow is secured by two alternative security measures. First of all, Amazon requires storing the app developer's app package name and certificate to create an API Key, which is used to identify the *C* app. Secondly, Amazon highly recommends the usage of the *tokenInfo* endpoint to avoid *RO* impersonation in the *Implicit* flow by verifying the authenticity of the `access_token` through a secure HTTP call to the *tokenInfo* endpoint.[5] Facebook supports a modified *Implicit* flow that secure the flow by requiring the interaction with their apps together with the storing of the app developer's package name and their certificate hash to verify the identity of the *C* app. In addition, Facebook enforces further security checks that can be activated from the app developer console, such as strict mode (avoid redirect hijacking by requiring exact match from OAuth redirect list), enforce HTTPS scheme registration, disable embedded OAuth login, app review process (in case the *C* app needs to access APIs and features that are not allowed by default, such as page public content access), and so on.

Regarding *PKCE* (6th column in Table 4.1), we discovered that among the 13 IdMPs that have a client SDK (i) 6 support *PKCE*, (ii) 2 optionally support *PKCE* (in case of Ping the *PKCE* query parameters are not automatically sent in the request as the app developer is required to set an optional field in the SDK; while for IBM the app developer can disable the *PKCE* support from the console), (iii) 2 do not support *PKCE* (Linkedin and Box) and violate BCP_{PKCE}, (iv) Facebook does not support *PKCE* due to the fact of using another grant flow rather than the traditional *Authorization Code*, and (v) 2 (Microsoft v1, HiDrive) do not support *PKCE* as they use different security measurement to protect the code interception. Microsoft v1 uses the package name and certificate to identify the *C* app, while HiDrive prompts the user to copy a code and paste it into the app to obtain the Authorization `code`.

Another problem in Microsoft v1 and Box SDK (5th column in Table 4.1) is the usage of an

[5] `https://developer.amazon.com/docs/login-with-amazon/impersonating-resource-owner.html`

embedded browser (against BCP_{UA}).

In addition, by reading the documentation, we have discovered that:

- Among the 14 IdMPs, only Google uses the AppAuth library as suggested SDK.
- Google and Auth0 are the only ones that provide documentation and an example for the proper implementation of the HTTPS scheme. Indeed, the initiation of associating the app with the HTTPS redirection depends on the proper implementation of the HTTPS scheme; otherwise, it will be susceptible to intent hijacking attack (a malicious app that registers the same HTTPS scheme as the benign app), which leads the user to choose for the right app [LWP+17]. The proper implementation of HTTPS scheme encompasses: (i) set the "`autoVerify`" filed as `TRUE`, (ii) create the valid `assetlinks.json` file format (package name, and app developer SHA fingerprint) and host it under the HTTPS server, and (iii) host the aforementioned file on the root directory of the server under "`.well-known`" path.
- Yahoo, BeeMinder, and DropBox do not refer to the native scenario on their documentation.
- Yahoo and BeeMinder do not provide the client SDK and do not mention OAuth/OIDC security BCPs for the native apps on their documentations.

Note that the alternative security measures which are mentioned above to secure the *Implicit* flow and avoid code interception attacks are related to the proprietary solutions. Thus, their integration in the BCPs documentation is difficult, as these proprietary solutions are firstly specific for an implementation pattern, and secondly, they are different from the reported BCPs by the OAuth WG and OIDF. Indeed, recommended BCPs are widely discussed within the OAuth community and related industries, and they are reliable and tested solutions [OMGMR+10]. Given that, native app developers are recommended to prioritize the adoption of BCPs rather than propitiatory security measures in the case of standard flow implementation.

4.2.3.2 Discussion on *R2*

We verified the compatibility of the selected IdMPs with AppAuth by integrating the AppAuth SDK within a demo app that aims to communicate with the selected IdMPs. As the AppAuth SDK enforces the OAuth/OIDC security BCPs described in [DB17], an IdMP compliance with OAuth/OIDC security BCPs makes the IdMP consequently compatible with the AppAuth SDK. We represent the analysis result in Table 4.2 and summarize the key points in the following.

As it is shown in Table 4.2, 7 IdMPs (Facebook, Beeminder, Box, Dropbox, Hidrive, Linkedin, Yahoo) are not compatible with the default configuration of AppAuth. For Facebook, Beeminder and HiDrive are due to the different grant types, in the case of Box, Dropbox, and Yahoo due

to the need to add `client_secret` and to remove *PKCE*, and for Linkedin due to the different `access_token` response format, which is missing the `token_type` field. The `token_type` field is required as mentioned in the OAuth Authorization Framework [Har12]. 7 IdMPs (Google, Amazon, Microsoft v2, OKTA, Auth0, Ping, IBM) are compatible with the default setting of AppAuth as they are following all the OAuth/OIDC security BCPs recommendations for native apps (Table 3.2). Note that, by using the AppAuth, the library enforces the BCPs by always using the *PKCE* query parameters in Ping and IBM and by avoiding using the *Implicit* flow for Auth0, Amazon, and Ping, and the *RO Password Credential* flow for IBM.

It is worth mentioning that it is possible to adopt AppAuth SDK to support Box, Dropbox and Yahoo (see Section 6.3). Although we need to modify the default behavior of the SDK (add `client_secret` and remove *PKCE*), we can still obtain some benefits in terms of security (e.g., the enforcing of external user agent instead of the embedded browser).

Discussion. Our updated analysis on 2021 shows a compliance improvement for Dropbox as they start to support *PKCE* solution (protection against T_{token}). Also, there is a minor change in the Box and Yahoo documentation for app developers. While for the Box, they provide a more detailed API reference. Yahoo deprecates its OpenID2 APIs and provides details for app developers on migrating to their new OpenID APIs.

Table 4.2: IdMPs Compatibility with AppAuth SDK.

IdMP Name	AppAuth Compatibility
Amazon	●
Auth0	●
Box	❍
BeeMinder	❍
Dropbox	❍
Facebook	❍
Google	●
HiDrive	❍
IBM	●
Linkedin	❍
Microsoft (v2)	●
OKTA	●
Ping	●
Yahoo	❍

❍ not compatible with the AppAuth SDK.
● compatible with the AppAuth SDK.

4.3 Analysis of Top-Ranked Google Play Store Apps

While in the previous section, we have presented the current status in terms of compliance with OAuth/OIDC security BCPs for native apps of most popular IdMPs, in this the section we focus on the client (*C*) side. Indeed, we consider *C* apps that integrate (at least) one of the IdMPs of Table 4.1, and we check how *C* apps implemented the OAuth/OIDC solutions. Our focus is both on OAuth/OIDC security BCPs compliance with the official SDK provided by the IdMPs and on the implementation level by checking how many *C* apps made insecure implementation choices that may lead to various threats as described in Section 3.2 and shown in Table 3.3. In addition, we also look at the relationship between BCPs compliance with the official SDK provided by IdMPs and insecure implementation choices within *C* apps: some implementation choices may turn a *fully compliant* solution provided by the IdMPs into a *not fully compliant*. We perform a preliminary compliance analysis on top-ranked Google Play Store apps downloaded in November 2019, and results are available in Appendix E. In the following, we report the analysis results performed on the latest version (February 2021) of the same set of apps. The main motivation to re-perform the compliance analysis is our interest in checking if there is an improvement in their solution compliance.

In Section 4.3.1, we present the selection criteria we followed to collect our dataset. In Section 4.3.2, we specify the features we collected from each *C* app. Section 4.3.3 presents the selection procedure we followed to find the proper static analysis tools. In Section 4.3.4, to analyze the collected features, we identify a set of research questions and for each research question, we report the results of our analysis.

4.3.1 Selection Criteria

For the apps to be representative, we selected them from top-rated apps in Google Play Store for 13 categories, which are more likely to use OAuth/OIDC solutions. We downloaded[6] 2505 top apps; the details about the total number of apps for each category and the procedure we used to detect them are provided in Appendix A. In the following, we detail the different phases (S1.-S4.) of the app selection procedure leading to the Resulting Dataset in 2019 (RD 2019), which consists of 10+ apps for each IdMP. Note that in our updated analysis, we do not follow the different phases of the app selection procedure through (S1.-S4.), as we are only interested in checking whether there is an improvement of security within the same set of apps. Thus, we report only the number of apps per each IdMP for the final Resulting Dataset in 2021 (RD 2021) by starting from the list of the apps in RD 2019 and checking if they still support the SSO/AD solutions.

[6] In November, 2019.

4.3.1.1 S1. Automatic Selection of *C* apps which (could) use IdMP

We wrote a python script that decompiles each app and it looks for a string equal to each IdMP endpoint in the decompiled files. We extracted each IdMP endpoint either from its documentation or from its developer console. The number of detected apps for each IdMP is provided in the second column of Table 4.3 (the details per each category are reported in Table A.1 in Appendix A). As several IdMPs may be integrated in the same app (e.g., an app supporting social login with both Facebook and Google), in the last two rows of Table 4.3 we report the total number of app instances ("Total Instances" of IdMPs) and the total number of apps ("Total Apps"), respectively. It is worth mentioning that for 6 IdMPs (marked in gray in Table 4.3) we do not find any apps. This is probably due to the fact that these IdMPs are providing enterprise OAuth/OIDC solutions with paid subscription fees; thus, apps supporting these IdMPs are not available on Google Play Store. Therefore, we do not consider these IdMPs in the next phases.

Table 4.3: Automatic and Manual selection (S1.-S2.), Static Analysis (S3.), Reaching 10 Apps (S4.) and Resulting Datasets (RD 2019, RD 2021).

IdMP	**S1.** # Detected Apps	**S2.** FPs	**S2.** Selected in S2 (i.e. #Selected S1 − FPs)	**S3.** Code not avail.	**S3.** Strongly obfuscated	**S3.** Premium not avail.	**S3.** Selected in S3 (i.e. Selected S2 − not avail.)	**S4.** # Similar Apps	**RD 2019** # Final Apps 2019	**RD 2021** # Final apps 2021
Amazon	25	22	3	0	0	0	3	7	10	7
Auth0	0	-	-	-	-	-	-	-	0	0
Box	25	13	12	0	0	1	11	-	11	11
BeeMinder	0	-	-	-	-	-	-	-	0	0
DropBox	17	0	17	1	0	1	15	-	15	15
Facebook	879	11*	10*	0	0	0	10	-	10	10
Google	22	12	10	2	1	2	7	3	10	10
HiDrive	0	-	-	-	-	-	-	-	0	0
IBM	0	-	-	-	-	-	-	-	0	0
Linkedin	10	4	6	1	0	0	5	5	10	10
Microsoft	21	9	12	1	0	0	11	-	11	10
OKTA	0	-	-	-	-	-	-	-	0	0
PING	0	-	-	-	-	-	-	-	0	0
Yahoo	13	4	9	0	0	0	9	1	10	10
Total Instances	**1012**	**77**	**79**	**5**	**1**	**4**	**71**	**16**	**87**	**83**
Total Apps	**921**	**72**	**49**	**5**	**1**	**2**	**46**	**16**	**60**	**57**

This reduces the number of considered IdMPs from 14 to 8.

4.3.1.2 S2. Manual selection of the apps using OAuth/OIDC solutions

After having reduced the number of selected apps by performing the automatic selection procedure described in S1., we have performed a manual selection by installing the apps on an emulator and executing them to detect whether they actually use the OAuth/OIDC endpoints of each IdMP. To do so, we have navigated each app through different screens (a.k.a. app activities), aiming to check if the navigated screen contains a button with the logo or the name of one of the IdMPs. Then, we clicked the button to check whether it was functioning or not. This procedure has been performed for all the app instances except for Facebook and premium app instances. While for the premium app instances we perform the selection procedure detailed in S3., in the case of Facebook we applied the following selection procedure. Being a manual selection for 879 apps unfeasible, among the list of apps already selected for other IdMPs, we have identified 8 apps which integrated Facebook as well. Furthermore, we have manually checked around 13 apps (one per category) to reach 10 suitable apps, indicated in Table 4.3 with an asterisk (*). As shown in the third column of Table 4.3, some of the apps selected in S1. were false positives (FPs). In most of the cases this is due to the fact that a library with the endpoint is included within the app but the library is used for another purpose. For the interested reader, more details about FPs are reported in Appendix A.2.2.

4.3.1.3 S3. Static Analysis of the Apps

Given our goal to extract from the dataset of apps the features we will list in Section 4.3.2, by performing static analysis, we further refined the set of selected apps. Indeed, not all the apps selected in S2. can be statically analyzed. As reported in Table 4.3, after a manual inspection in S3., we identified three reasons. In most of the cases (5 apps), the relevant code of the apps is not available in the *APK*, because it is dynamically loaded. In one case, the app is strongly obfuscated, meaning that the methods and the parameters were renamed. This made our static source code analysis unfeasible. Finally, in another case (4 apps), the full functionalities were not included in the selected app, but in a paid version of the app, available in the Google Play Store.

4.3.1.4 S4. Reaching 10 *C* apps per IdMP by manually executing "similar apps"

For some IdMPs we did not find at least 10 app instances. Thus, for a randomly chosen app (among the ones selected in S2.) per each IdMP, we performed a manual search in the "Similar Apps" section of Google Play Store, so to find similar apps that could be possible candidates. The rationale behind this choice is that similar apps provide similar functionalities and should

use similar SSO/AD mechanisms, and thus it should be easier to find other apps integrating the IdMPs. We installed and executed the apps in the "Similar Apps" till reaching our goal of 10, and we repeated S2. and S3. In this phase, as reported in column S4. of Table 4.3, we selected 16 apps. (8 of them are among the top-rank apps.)

4.3.1.5 Resulting Dataset

We started from an initial set of 2505 apps, and we applied the previous phases (S1. through S4) to obtain the final dataset of apps integrating (at least) one of the IdMPs of Table 4.1. As shown in Table 4.3, the resulting dataset in November 2019 includes 60 apps and 87 instances. While for the updated analysis in February 2021, the final resulting dataset includes 57 apps and 83 instances. The reason behind the number decrease is related to the fact that 4 apps (3 Amazon, 1 Microsoft) removed their support for the SSO and/or AD scenario. Note that the reported number in the resulting data set (Table 4.3) does not necessarily mean that only 83 app instances among 2505 apps are integrating SSO and/or AD scenarios. As it is reported in Table 4.3, we identify around 879 apps that contain the Facebook endpoint. However, we are interested in having only 10 app instance to be representative for Facebook. Thus, we have not considered all the identified Facebook apps in the S1..

4.3.2 Selected Features

To analyze the security status of the OAuth and OIDC implementations, for each app instance of OAuth/OIDC solution—selected using the procedure reported in Section 4.3.1—we have extracted the following features:

Scenario: states if the app instance consists of either a *SSO* Login or *AD* scenario or both.

SDK support: specifies whether the app uses official SDK (including AppAuth) or unofficial/no SDK. By official SDK, we mean that the app integrates the SDK suggested in the documentation by the IdMP (cf. Table 4.1). By unofficial/no SDK we mean that the app either uses an SDK, which is not suggested by the IdMP, or it builds the OAuth/OIDC flow manually.

OAuth/OIDC security BCPs compliance: to check how many app instances are *fully compliant* with OAuth/OIDC security BCPs, we referred to the features summarized in Table 3.2. It is worth mentioning that the iGov and FAPI profiles security BCPs are out of scope of our analysis. This is due to the following reasons: (i) these BCPs are applied for the public administration and financial sector services apps; and (ii) these apps do not use the social login providers to authenticate users and/or authorize the access to user's resources. We have thus collected information about (i) *Flow*, (ii) *UA*, and (iii) *PKCE.*

Implementation issues: to check how many app instances contain implementation issues we have collected information about:

- *HTTPS scheme configuration:* presence of the field "`autoVerify=TRUE`" in the HTTPS scheme declaration; the lack of this field results in a wrong verification process of the app identity.
- *Secret Management:* presence of the `client_secret` in plaintext within apps.

For space constraint, in Table 4.4 we report an aggregated view of the features per IdMP. The complete extracted features are reported in the companion website.

4.3.3 Selected Static Analysis Tools

To extract the features mentioned above in the previous section from the selected apps, we performed a manual inspection of the code. We used the support of the following two open-source static analysis tools to perform our OAuth/OIDC security BCPs compliance analysis within apps: JADX, and SUPER Android Analyzer [JAD18, Sup18]. More details about how we used these tools to extract the selected features are reported in Appendix B.

Table 4.4: Overview of the Selected Features per IdMP.

IdMP	**Scenario**		**SDK Support**			**BCPs**			**Impl. fault**	
	# SSO	#AD	# official SDK	# unofficial SDK	# no SDK	# code without secret	# external UA	# *PKCE*	# HTTPS misconf.	# plain secret
Amazon	4	3	4	0	3	6	6	4	0	NA
Box	0	11	6	1	4	0	0	0	1	4
DropBox	1	14	10	1	4	0	11	7	NA	5
Facebook	10	0	10	0	0	NA	5	NA	NA	NA
Google	4	6	0	2	8	6	4	0	NA	3
Linkedin	10	0	0	1	9	0	0	0	NA	6
Microsoft	2	8	5	1	4	8	3	2	NA	1
Yahoo	10	0	NA	0	10	0	1	0	NA	7
Total Instances	41	42	35	6	42	20	30	13	1	26

To find the proper tools for this purpose, we performed a comprehensive analysis on available static analysis tools in the wild. We analyzed 240+ source-code analysis tools, which are gathered from the research papers [SH14, STTPLB14, AC13, SBGM16] (to name only few of them) and the Android security analysis tools repository [Bha18]. Furthermore, we checked the tools for their capabilities such as bug detection, and Web View detection. In particular, we have that:

- 168 out of 247 tools are capable of performing varied security analysis;
- Among the 168 security analysis tools, only 57 of them are open-source;
- Only 27 out of 57 open source tools are available online.

We list and report the final 27 tools and their capabilities in Table B.1, which is available in Appendix B.

4.3.4 Research Questions and Results

To analyze the selected features, we identified the following research questions:

R1. How many *C* apps are violating OAuth/OIDC security BCPs?

R2. Focusing on app instances using official SDK. How many app instances are violating OAuth/OIDC security BCPs? The violation of these security BCPs is due to IdMP or *C* developer choices?

R3. Focusing on app instances using unofficial or no SDK. How many app instances are violating OAuth/OIDC security BCPs?

R4. How many app instances have HTTPS scheme misconfiguration or `client_secret` in plaintext?

In the following, we discuss each research question.

4.3.4.1 Discussion on *R1*

To answer this question, per app instance we have checked if they are following OAuth/OIDC security BCPs recommendations of Table 3.2, namely:

1. the *Flow* is the *Authorization Code* flow (without `client_secret`);

2. the *UA* is an external browser;

3. The *PKCE* solution is implemented.

In Table 4.5, we report the results per IdMP by specifying OAuth/OIDC security BCPs coverage of each app instance belonging to our Resulting Dataset, where: with *x*/3 we mean that the app instance is following *x* of the aforementioned recommendations. We call *fully compliant* with OAuth/OIDC security BCPs only app instances with a score of 3/3. From the results, we have that:

- 5/83 app instances are *fully compliant*: 3 for Amazon and 2 for Microsoft;
- all the app instances for Box, Dropbox, Facebook, Google, Linkedin and Yahoo are *not fully compliant* with OAuth/OIDC security BCPs.

Considering the number of *C* apps (instead of the app instances), we have that 53/57 (93%) of them are *not fully compliant* with OAuth/OIDC security BCPs. Table 4.6 reports the results per IdMP by specifying OAuth/OIDC security BCPs violation of each app instance belonging to our Resulting Dataset. From the results, we have that:

- native app developer of Box and Amazon app instances have been failed in adoption of all OAuth/OIDC security BCPs;
- BCP_{flow} is the most overlooked OAuth/OIDC security BCPs by native app developer.

Table 4.5: Security BCPs Coverage per IdMP.

IdMP	**Sec. BCPs Coverage** 0/3	1/3	2/3	3/3	**% of app instances not fully compliant**
Amazon	0	1	3	3	57%
Box	10	1	0	0	100%
Dropbox	4	4	7	0	100%
Facebook	5	5	0	0	100%
Google	4	2	4	0	100%
Linkedin	10	0	0	0	100%
Microsoft	2	5	1	2	80%
Yahoo	9	1	0	0	100%
Total Instances	44	19	15	5	94% (78/83)

4.3.4.2 Discussion on *R2*

From Table 4.4, we know that 35/83 app instances use the official SDK. Out of these, only 5/35 (3 for Amazon and 2 for Microsoft) are *fully compliant* with OAuth/OIDC security BCPs.

For the remaining 30 app instances *not fully compliant*, to understand if the violation of OAuth/OIDC security BCPs is due to the SDK officially provided by the IdMP or is introduced by the *C* developer, we have compared the features gathered from this analysis with Table 4.1. In addition, by considering the possible threats from Table 3.3, we have that:

- 2 app instances (1 Amazon, 1 Microsoft) are *not fully compliant* due to app developers choices: they remove the support to *PKCE* which is mandatory for native apps. In the case there are not any additional security mechanisms, app developers would be vulnerable to token leakage (T_{token}).
- 15 app instances (6 Box, 7 Dropbox, 2 Microsoft) are *not fully compliant* due to the IdMP. 8 app instances (6 Box, 2 Microsoft) do not support *PKCE*, which may lead to T_{token} in lack of additional security mechanisms. 13 app instances (6 Box, 7 Dropbox) support the *Authorization Code* flow with `client_secret` that may lead to client impersonation (T_{secret}) if the `client_secret` parameter is stored in plaintext within the *C* app. Finally, Microsoft supports an embedded browser that is highly discouraged within third-party apps, as its usage can lead to leak of user credentials (T_{cred}).
- 13 app instances (10 Facebook, 3 Dropbox) are *not fully compliant* due to both IdMP and app developers. The app developer fault in case of Facebook is using an embedded browser instead of external that could make app instances susceptible to T_{cred} and in case of Dropbox is lack of PKCE support that could make app instances vulnerable to T_{token}.

Table 4.6: Overlooked security BCPs per IdMP by native app developers.

IdMP	**Sec. BCPs Violation**		
	BCP_{flow}	BCP_{UA}	BCP_{PKCE}
Amazon	1	1	2
Box	11	11	11
Dropbox	15	4	8
Facebook	10	5	N/A
Google	4	6	10
Linkedin	10	10	10
Microsoft	2	7	8
Yahoo	10	9	10
Total	63	53	59

While the IdMP violation is related to the flow (*Implicit* for Facebook and use of the `client_secret` for Dropbox). While in the case of Facebook the usage of *Implicit* flow is not a problem as they implement additional security mechanisms to avoid T_{token}, in case of Dropbox, the usage of the `client_secret` (in plaintext) may lead to T_{secret}.

In addition, we can observe that:

- $\sim$ 87% (26/30) are using a flow different from the *Authorization Code* flow without secret (10 *Implicit*, 16 *Authorization Code* with secret).
- $\sim$ 43% (13/30) are using an embedded browser.
- $\sim$ 43% (13/30) are omitting *PKCE* within their implementation.

4.3.4.3 Discussion on *R3*

From Table 4.4, we know that 48/83 app instances use the unofficial or no SDK. In particular, we have:

- 10 app instances are not using an official SDK as is not provided by the IdMP [10/10 Yahoo].
- 32 app instances have manually integrated OAuth/OIDC even if there is an official SDK for the IdMP [3/7 Amazon, 4/11 Box, 4/15 Dropbox, 8/10 Google, 9/10 Linkedin and 4/10 Microsoft].
- 6 app instances are using an unofficial SDK even if there is an official SDK for the IdMP [1/11 Box, 1/15 Dropbox, 2/10 Google, 1/10 Linkedin and 1/10 Microsoft].

To answer *R3*, per app instance we have checked OAuth/OIDC security BCPs recommendations (see Table 3.2). We have that all the app instances ($\sim$ 100%) are *not fully compliant*. Out of these, we can observe that:

- $\sim$ 77% (37/48) are using a flow different from *Authorization Code* without `client_secret`.
- $\sim$ 83% (40/48) are using an embedded browser.
- $\sim$ 96% (46/48) are omitting *PKCE* within their implementation.

4.3.4.4 Discussion on *R4*

Analyzing *R2*, we have discovered that 2 app instances are *not fully compliant* with OAuth/OIDC security BCPs due to app developers modification of the official SDK code (using an embedded browser instead of an external one). In this research question, we keep the app developer perspective but we focus on two implementation aspects: the HTTPS scheme configuration and the storing of the `client_secret`.

As shown in Table 4.4, we have that:

- only 2 app instances are using the HTTPS scheme; out of these, 1 does not specify the field "`autoVerify=TRUE`" in the HTTPS scheme declaration; the lack of this field results in a wrong verification process of the app identity and in the absence of additional security mechanisms would lead to T_{code} and then T_{token}.
- 52 app instances are sending the `client_secret`; out of these, 26 store the secret in plaintext within apps that makes *C* apps susceptible to T_{secret}.

The problem of using `client_secret` in native apps is due to its nature. By definition, being a public client, a native app is not able to keep any value confidential. So OAuth/OIDC solutions should not base the client authentication on this value, especially if it is stored in plaintext. However, as we have previously shown many IdMPs are still requiring the sending of the secret. If the secret is necessary for backward compatibility, in order not to compromise the web scenario, a good practice is to use different `client_secret` value for the native scenario. For the 52 app instances sending the `client_secret`, we have that only 16 app instances are using different `client_secret` values for the web scenario (4 Google, 2 Microsoft, 10 Yahoo).

4.3.4.5 Summary

- $\sim$ 86% (30/35) of the app instances using official SDK are *not fully compliant* with OAuth/OIDC security BCPs;
- $\sim$ 100% (48/48) of the app instances using unofficial/no SDK are *not fully compliant* with OAuth/OIDC security BCPs;

Discussion. Our updated analysis in 2021 represents an improvement towards app compliance compared to the preliminary analysis, which is conducted in November 2019. The main differences are summarized as follows:

- 3 apps removed their Amazon support for SSO;

- 8 apps (1 Amazon and 7 Dropbox) implement *PKCE* that helps to reduce the possibility of T_{token}; and
- 8 apps (1 Amazon, 2 Dropbox, 5 Facebook) implement external *UA* instead of embedded browser (mitigating T_{cred}).

The improvement mentioned above can be linked to the latest OAuth security BCPs draft [LBLF21] that provides more specific recommendations. Also, we contacted app developers of 57 apps (October 2020) to inform them about our findings w.r.t. their solution compliance. 4 out of 57 app developers ($\sim 7\%$) have replied to our report. While 2 of them are working on our findings, one of them (Cisco Webex) patches the reported problem and acknowledges us in their public release note. [7] Furthermore, one app developer does not consider a plan to fix the reported issues. However, in our latest analysis, we notice that 15 apps address some of our findings without acknowledging our report.

It is worth mentioning that non-compliance with OAuth/OIDC security BCPs does not necessarily mean that their solution is vulnerable to the reported attacks. However, their compliance with OAuth/OIDC security BCPs can decrease the attack surface. To represent the feasibility of exploiting the reported threats, we perform the client impersonation attack on some non-compliant apps.[8] For other attack examples, the reader can refer to [ARFT19, WZL+15, CPT+16, NS19], where researchers presented successful attacks due to the lack of proper OAuth/OIDC security BCPs countermeasure.

[7] https://st.fbk.eu/news/2021/03/26/bug-reported-to-cisco/

[8] https://cutt.ly/YzMxLPC

Chapter 5

OAuth/OIDC Risk Assessment

This chapter presents a reference model for OAuth/OIDC deployments to capture all the relevant features w.r.t. their implementations and then formalize it as an optimization problem that can be solved by using available risk analysis tools; aiming at minimizing the security risks within OAuth/OIDC implementations and helping native app developers to make an informed decision concerning their implementation choices.[1]

Below, we first motivate the need for a risk assessment and its complexity in the context of OAuth/OIDC solutions. Then we introduce a reference model for OAuth/OIDC implementations extending the obtained information in Chapter 3 to assist native app developers in performing an OAuth/OIDC risk assessment.

5.1 Context and Motivations

Risk assessment plays a key role in identifying, estimating and prioritizing potential threats in the app development process. Indeed, conducting risk assessment helps native app developers know how vulnerable their current implementation is and plan the required mitigation through overlooked implementation choices. As mentioned in Chapter 3, the OAuth WG and the OIDF have published a set of BCPs to assist native app developers with securing their implementations against possible threats. However, our analysis in Chapter 4 reveals that, unfortunately, many apps under analysis still do not follow these BCPs. One of the possible reasons is that native app developers are usually focus on providing the new functionalities for the app, and they may not have enough resources to perform a risk assessment procedure to be able to compare the risk level poses by selecting different implementation choices rather than the one already implemented within their apps. Indeed, this task can be even more difficult and complex in the context of

[1] This chapter is based on our published work in [DSCR21].

OAuth/OIDC solutions due to the following reasons:

- many choices of IdMPs with various configuration options;
- various implementation options for native app developers to implement within their apps;
- a maze of documents and guidelines to be considered by native app developers to perform a comprehensive and flawless risk assessment, which is challenging for non-security experts;
- native app developers need to be aware of which BCPs to follow in order to meet the security requirements of their apps;
- native app developers need to be aware of potential threats concerning the OAuth/OIDC implementations and their interrelations to be able to correctly model the risk propagation concerning their implementations properly.

The difficulties mentioned above motivated us to provide a reference model for OAuth/OIDC deployments aiming to assist native app developers in performing a risk assessment. In particular, we use the reference model to formalize risk assessment problem P_{risk} as an optimization problem, namely a security risk assessment and trade-off analysis of OAuth/OIDC implementations. We consider the perspective of native app developers within P_{risk}, taking into account the propagation of the negative effects of the threats and the positive effects of the mitigation offered by implementing the relevant security measures corresponding to the selected implementation options.

We preliminary provide (in Section 5.2) a reference model to capture all the relevant features w.r.t. OAuth/OIDC solutions implementations. We then formalize P_{risk} (in Section 5.3), paving the way for a tool-based approach for risk assessment.

5.2 OAuth/OIDC Reference Model

We use the reported information in Chapter 3 as a foundation and extend them to build a reference model applicable for the risk assessment purpose. In Chapter 3 we presented the relation among threats and BCPs and how these BCPs are intended to thwart threats. In this section, we analyze the OAuth/OIDC solutions implementations from a risk analysis perspective, which demands to:

- Extract all security/privacy related implementation choices that OAuth WG and OIDF have introduced for OAuth/OIDC implementations. In Chapter 3, we focused on the implementation choices prescribed by BCPs that we referred to them as query parameters, while in this chapter, we consider all available implementation choices for query parameters to

assess the risk level, in case the BCPs are not (completely) followed. We call the implementation choices, feature. To have a comprehensive reference model, we not only need to consider the authorization and token request features in the authorization flow (discussed in chapter 3), but also we need to consider the client-specific implementation-related task

Table 5.1: OAuth/OIDC Reference Model.

Deployment Place	Atomic Features	Threat (T)	Goal (G)	Consecutive T	Consecutive G	PL	LL
		Security Feature					
Authorization Request	code	T_{token}	PD Conf	-	-	3	3
	token					1	
	client_credentials					3	
	password					1	
	hybrid					3	
	nonce	T_{token} , $T_{session}$	PD Conf	-	-	5	5
	state	$T_{session}$	PD Conf	-	-	5	5
	request^{c,hyd}	T_{code}	AT Conf	T_{token}	PD Conf	5	5
	request_uri^{c,hyd}					5	
	query^{c,hyd}					1	
	request^{i,pswd}	T_{token}	PD Conf	-	-	3	5
	request_uri^{i,pswd}					3	
	query^{i,pswd}					1	
	form_post^{c,hyd}	T_{code}	AT Conf	T_{token}	PD Conf	5	5
	fragment^{c,hyd}					1	
	query^{c,hyd}					1	
	form_post^{i}	T_{token}	PD Conf	-	-	5	5
	fragment^{i}					1	
	code challenge	T_{token}, $T_{session}$	PD Conf	-	-	5	5
	Plain	T_{token} , $T_{session}$	PD Conf	-	-	1	5
	S256					5	
Token Request	mTLS	T_{token}	PD Conf	-	-	3	3
	private key jwt					3	
	client_secret_jwt					2	
	client_secret_basic					2	
	cleint_secret_post					2	
	code_verifier	T_{token}, $T_{session}$	PD Conf	-	-	5	5
Authorization/Token Request	full redirect_uri^{idmp,c,hyd}	T_{code}	AT Conf	T_{token}	PD Conf	3	3
	pattern redirect_uri^{idmp,c,hyd}					1	
	full redirect_uri^{idmp,i}	T_{token}	PD Conf	-	-	2	2
	pattern redirect_uri^{idmp,i}					1	
Header	binding IdMP metadata^{cl,c,hyd}	T_{code}	AT Conf	T_{token}	PD Conf	2	2
	binding IdMP metadata^{cl,i}	T_{token}	PD Conf	-	-	2	2
	referrer^{cl,c,hyd}	T_{code}	AT Conf	T_{token}	PD Conf	3	3
	referrer^{cl,i}	T_{token}	PD Conf	-	-	2	2
Console	distinct redirect uri^{cl,c,hyd}	T_{code}	AT Conf	T_{token}	PD Conf	5	5
	distinct redirect uri^{cl,i}	T_{token}	PD Conf	-	-	2	2
Client	open redirect validation^{cl,c,hyd}	T_{code}	AT Conf	T_{token}	PD Conf	5	5
	open redirect validation^{cl,i}	T_{token}	PD Conf	-	-	2	2
	state Validation^{cl,c,hyd}	T_{code}	PD Conf	-	-	5	5
	state validation^{cl,i}	T_{token}	PD Conf	-	-	5	5
	storing IdMP metadata^{cl,c,hyd}	T_{code}	AT Conf	T_{token}	PD Conf	5	5
	storing IdMP metadata^{cl,i}	T_{token}	PD Conf	-	-	5	5
	issuer validation^{cl}	T_{code}	AT Conf	T_{token}	PD Conf	5	5
	id token validation^{cl}	Impersonation	PD Conf	-	-	5	5
		Privacy Feature					
Authorization Request	claims	Comp data mini	Data mini	Noncompliance	Compliance	5	5
	scope	Comp data mini	Data mini	Noncompliance	Compliance	5	5
	purpose	Comp Pur spec, Trans	Pur spec, Trans	Noncompliance	Compliance	5	5
	verified_claims	Comp PD accu	PD accu	Noncompliance	Compliance	5	
	vot	Impersonation, Comp PD accu	PD accu, PD Conf	Noncompliance	Compliance	5	5
	acr_values					5	
	login	Comp Transparency	Transparency	-	-	3	3
	select_account					3	
	consent					3	
console	pairwise	Linkability	Unlinkability	-	-	5	5
	public					1	

LEGEND. idmp: idmp only feature, cl: client only feature, c: code, i: implicit, hyd: hybrid, pswd: password comp: compromise, PD: personal data, AT: access token, trans: transparency, data mini: data minimization, conf: confidentiality, pur spec: purpose specification, PL: protection level, LL: likelihood level.

(e.g., `state validation`), that is, features that app developers need to implement and checks they need to perform. To illustrate, let us consider the OAuth/OIDC *Response type* feature that determines the authorization flow to be used within the OAuth/OIDC implementations. While the BCPs recommend the usage of `code` for the *Response type* feature (BCP_{flow} in Table 3.2), in this chapter, we take into account all the possible values for the aforementioned feature, namely: `code`,`token`,`password`,`client_credentials`,`hybrid`.

- Identify and group the features that the OAuth WG and OIDF that have been introduced for a common purpose. To illustrate consider the *Client authentication* feature. All the members of the aforementioned feature have a common purpose of providing a way to authenticate *C* app at the Token endpoint of *AS*. This is an important step as it identifies all the interchangeable features in the OAuth/OIDC solutions implementations.

- Identify security/privacy threats to the features. As it is already mentioned in Chapter 3, in the OAuth context, threats are defined as any intentional attacks on OAuth tokens and resources protected by them. Thus, we considered the list of threats in [LMH13, LBLF21] to extend security and privacy threat introduced in Section 3.2, and Section 3.3.4, respectively. In particular, we add the *Impersonation* and *Noncompliance* threats. The reason for our choice is due to fact that the aforementioned documents structure the implementation features along the lines of the protocol structure. Thus, it assists native app developers in implementing each part of the protocol securely, for example, all implementation features to avoid the threat of obtaining authorization code (T_{code}). Note that the extracted information may serve as a foundation and native app developers may detail and extend the list of identified security/privacy threats to account for the specific properties of their deployment.

- Identify security/privacy goals, consecutive threats and goals for each security/privacy-related features. In Chapter 3, the focus was on the relation among BCPs and threats and how these BCPs are intended to thwart threats; thus, the consideration of a single set of threats as reported in Section 3.2 was sufficient for the aforementioned purpose. However, in this chapter, in order to be able to model the risk propagation due to the interrelations among identified threats, we introduce the consecutive threats and goals, which will be further detailed in the rest of this section.

- Assign likelihood/protection level to each security/privacy features. While likelihood level represents how likely the system is posing a threat, the protection level provides the level of protection for each feature against threats. The aforementioned values are necessary for calculation of the risk level in a risk assessment procedure.

Given that, we create a reference model for OAuth/OIDC solutions as reported in Table 5.1 with the following definitions:

Deployment place represents where the security/privacy features are deployed during the OAuth/OIDC solutions implementation. As reported in Table 5.1, it can be either of the following values:

- Authorization request. categorizes the security/privacy features which are used in Step 1 in Figure 3.2.
- Token request. categorizes the security/privacy features which are used in Step 4 in Figure 3.2.
- Authorization/Token request. categorizes the security/privacy features which are used in both Step 1 and Step 4 in Figure 3.2.
- Header. categorizes the features, which are set either in authorization or *UA* header.
- IdMP's Console. refers to the features which are set by app developers during the registration phase at IdMPs.
- Client. refers to specific implementations tasks or checks that app developers need to implement or perform within their apps.

Atomic features represents the extracted security and privacy features from OAuth and OIDC security-related documents. Each atomic features (herafter, *af*) represents either: (i) OAuth/OIDC query parameter name (e.g., `nonce`), (ii) OAuth/OIDC query parameter value (e.g., `SHA256`), (iii) OAuth/OIDC functionality aspects (e.g., `full redirect_uri`), or (iv) client specific implementation-related tasks (e.g., `id token validation`). A detailed definition for each *af* in the reported categories is provided in Table 5.2.

Threat (T) represents any hostile actions that aim to damage security and/or privacy assets, which are OAuth tokens and/or user resources in the OAuth/OIDC context. For each *af*, we identify the relevant threat(s) by extending the reported threats in Sections 3.2 and 3.3.4 to cover all the potential threats. In particular, we add the following two threats, namely: *Impersonation* and *Noncompliance*. While the former represents a scenario that an attacker is tricking a client into accepting a manipulated `id_token` to impersonate an user, the latter represents a case where the privacy requirements do not meet by the client in question.

Goal (G) represents the security/privacy goals that we aim to protect against threats. For each threat *T*, we identified its relevant security or privacy goals. While for the security goals, we consider both OAuth tokens and user resources as our assets and define the *access token confidentiality* (AT Conf), and *personal data confidentiality* (PD Conf) based upon it. In the case of privacy, the identified goals are mainly focused on the user resources (e.g., personal data) to satisfy the requirements of user privacy. Therefore, we extended the identified privacy goals that are reported in Section 3.3.4 with *compliance* goal to cover the newly defined *noncompliance* privacy threat.

Consecutive T represents a circumstance that a threat *T* could expose the client to another threat.

Consecutive G represents the goal that can be compromised in the case of Consecutive T.

Protection Level (PL) represents the level of protection that each *af* provides, and it is calculated based on the level of contribution to mitigating the relevant threat. The protection level value ranges from 1 (baseline) to 5 (highest). To illustrate, let us consider *af*s `mTLS` and `client_secret_basic`. We assign *af*s `mTLs`, `client_secret_basic` the protection level of 3, and 2, respectively. The reason for this assignment is due to the fact that for the asymmetric client authentication methods (`mTLS` and `private_key_jwt`), *AS* does not need to store sensitive symmetric keys (`client_secret_basic`, `client_secret_post`, and `client_secret_jwt`), making these methods more robust in comparison with symmetric client authentication methods [LBLF21].

Likelihood Level (LL) represents how likely the system is posing a threat if the related *af*(s) is (are) not implemented. The likelihood level similar to the protection level is between 1 (less likely) to 5 (most likely). To assign the likelihood values, while for the case of posing a threat by a single *af*, we consider the exact value of protection level as the likelihood level. In the case of posing a threat if non of related *af*s are implemented, we consider the maximum value of protection level among the related *af*s as a likelihood level.

To achieve an appropriate level of security and privacy in the OAuth/OIDC deployments, the OAuth WG and the OIDF have introduced different *af*s for a common functionality within the OAuth/OIDC deployments. These *af*s are interchangeable and they provide a different level of protection for the relevant threat. We group together such *af*s, and call them *composed feature* (*cf*), which are represented in Table 5.3. While an IdMP can support either one or all the *af*s belong to a *cf*, a client can implement only one *af* from a *cf*, per request. It is worth mentioning that for *cf*s, the likelihood level is assigned to the *cf*. That is because the related threat(s) only happen if the app developer does not implement any *af*s of a *cf*.

To illustrate the concepts presented in Table 5.1, consider the *cf Request*. It comprises of *af*s `request`, `request_uri`, and `query`. They are introduced to meet the goal *access token confidentiality*, by protecting authorization request against the threat T_{code}. In addition, the threat T_{code} can lead to consecutive threat T_{token}, which is related to the consecutive goal *personal data confidentiality*. We assign the protection level of 5 to *af*s `request` and `request_uri`, as they pass OAuth/OIDC requests query parameters in a signed and optionally encrypted single, self-contained query parameter manner [SBJ+14] that provide the request integrity and confidentiality benefits. In the case of *af* `query`, we assign the protection level of 1 as it adds the authorization request query parameters directly in the authorization requests, which does not provide request integrity and confidentiality in comparison with *af*s `request` and `request_uri`. This is an important consideration, as it allows controllers to make an informed decision on the IdMP s/he chooses (not all IdMPs support all the *af*s), or/and the *af*s they decide to implement.

The reference model aims to pave the way for a tool-based approach for risk assessment. Therefore, app developers may use our reported reference model as the baseline and extend the refer-

ence model by adding *af*s which are specific for their client implementations. Also, they may update the protection/likelihood levels for the *af*s based on their client operating domain.

Table 5.2: Atomic Features Definitions.

Atomic Features	Description	Source
`code,token,client_credntials, password,hybrid`	Defined values for the response_type OAuth/OIDC feature to determine the authorization flow to be used.	[SBJ+14]
`state`	OAuth/OIDC feature with an opaque value used to maintain the state between the authorization request and response.	[SBJ+14]
`nonce`	OIDC feature with an opaque value used to associate a client session with an `id_token`.	[SBJ+14]
`request`	OIDC feature for sending the authorization request query parameters in a single query parameter and to be optionally signed.	[SBJ+14]
`request_uri`	OIDC feature to enable authorization requests to be passed by reference. The object value retrieved from the resources at the specified URL.	[SBJ+14]
`query`	OAuth/OIDC default functionality to serialize the authorization request parameters which add them directly in the request.	[SBJ+14]
`form_post,fragment,query`	Defined values for the response_mode OAuth/OIDC feature to inform the IdMP about the mechanism for returning query parameters from the *AS* endpoint.	[SBJ+14]
`plain,S256`	Defined values for the code_challenge_method OAuth/OIDC feature to define the transformation method used to calculate the code_challenge.	[SBA15]
`code challenge`	OAuth/OIDC feature that represents a challenge derived from the code_verifier.	[SBA15]
`client_secret_jwt,private_key_jwt ,mTLS,client_secret_basic,client_ secret_post`	A set of client authentication methods that are introduced by OAuth/OIDC to authenticate the clients at the Token endpoint.	[SBJ+14, CBSL19]
`code_verifier`	OAuth/OIDC feature that contains a high-entropy cryptographically created string that is unique for each authorization request.	[SBA15]
`full redirect_uri,pattern redirect_uri`	Defines the redirection URL registration policy within the IdMP developer console.	[LBLF21]
`binding IdMP metadata`	App developers must bind IdMP metadata for each authorization request which are sent to *AS* by setting them in *UA* header.	[LBLF21]
`referrer`	App developers must set the Referrer-Policy HTTP header into `no-referrer`.	[LBLF21]
`distinct redirect_uri`	App developers must register distinct redirection URL per each IdMP.	[LBLF21]
`open redirect validation`	App developers must avoid the acceptance of arbitrary value by the registered redirection URL of the app and creates a redirect to it.	[LBLF21]
`state validation`	App developers must validate the equality of the `state` feature value in the authorization request and response.	[Har12]
`storing IdMP metadata`	App developers must store the IdMP metadata that they send the request and check if the request was received from the correct IdMP.	[LBLF21]
`issuer validation`	App developers must verify the equality of the returned issuer value with the IdMP that they communicate.	[LBLF21]
`id token validation`	App developers must verify all the claims within the `id_token` by following the 13 recommendations within OIDC document.	[SBJ+14]
`claims`	OIDC feature to enable requesting individual claims to be returned from the *Userinfo* and/or in `id_token`.	[SBJ+14]
`scope`	OAuth/OIDC feature that contains string values to specify the scope of the `access_token`.	[SBJ+14]
`purpose`	OIDC feature to enable stating the purpose of asking the specific claim by the client.	[LF20]
`verified_claims`	OIDC feature that contains object or array of one or more verified claims.	[LF20]
`vot,acr_values`	OAuth/OIDC feature to request IdMP for guaranteeing a certain level of identity proofing.	[SBJ+14, RJ18]
`login,select_account,consent`	Defined values for the prompt OAuth/OIDC feature to specify whether IdMPs prompt users for re-authentication and consent.	[SBJ+14]
`pairwise,public`	Defined values for the subject_type OIDC feature to determine how the app unique identifier is generated by the IdP.	[SBJ+14]

5.3 Problem definition

In this section, we formalize the identified OAuth/OIDC risk assessment problem, namely P_{risk} considering the reference model reported in Table 5.1.

Let $\mathcal{AF}$ be the set of *af*s associated with an OAuth/OIDC deployment shown in the second column of Table 5.1. We split the set $\mathcal{AF}$ in three disjoint subsets: $\mathcal{AF} = \mathcal{AF}_{common} \cup \mathcal{AF}_{idmp} \cup \mathcal{AF}_{cl}$.

The set $\mathcal{AF}_{common}$ includes the *af*s clients cannot implement unless the IdMP support them, like `nonce`. An IdMP can support more than one *af* from a *cf*, while the client can implement only one for a given request. The set $\mathcal{AF}_{idmp}$ includes *af*s that IdMPs need to enforce (marked with the "idmp" superscript in Table 5.1); clients need only to adopt them, like `pattern redirect_uri`. The set $\mathcal{AF}_{cl}$ includes the *af*s that clients need to implement and checks that they need to perform (marked with the "cl" superscript in Table 5.1), like `issuer validation`. Let $support_{idmp}$ be a Boolean mapping from the set $\mathcal{AF}_{common} \cup \mathcal{AF}_{idmp}$ for *idmp* ranging over a given set of IdMPs. An atomic feature *af* maps to a true value *iff idmp* supports *af*. Similarly, let $implement_{cl}$ be a Boolean mapping from the set $\mathcal{AF}_{common} \cup \mathcal{AF}_{cl}$ for a given client *cl*. An atomic feature *af* maps to a true value *iff cl* implements *af*.

Notice that, for $af \in \mathcal{AF}_{\text{common}}$, we say that $implement_{cl}$ is *constrained* by $support_{idmp}$, meaning that $implement_{cl}$ is strictly dependent on $support_{idmp}$, i.e. $af \in \mathcal{AF}_{\text{common}}$ can map to true only if it maps to true in $support_{idmp}$. Indeed, the entity responsible to implement the client can decide to implement or not some atomic features among the one supported by *idmp*, but in any case it cannot implement the atomic features that are not supported by *idmp*.

Problem P_{risk} definition. Let $\mathcal{CF}^* \subset \mathbf{P}(\mathcal{AF})$ (where **P** stands for the power set) be a set of sets of *af*s, including the *cf*s defined in Table 5.3 as well as a set $\{af\}$ for each $af \in \mathcal{AF}$ that does not belong to any composed feature in Table 5.3. Thus, $\mathcal{CF}^* = \{\ldots, \{\texttt{nonce}\}, \{\texttt{state}\}, \{\texttt{request}, \texttt{request\_uri}, \texttt{query}\}, \ldots\}$. Note that all the sets $S \in \mathcal{CF}^*$ are pairwise disjoint and $\bigcup_{S \in \mathcal{CF}^*} S = \mathcal{AF}$, that is $\mathcal{CF}^*$ is a partition of $\mathcal{AF}$. Let $\mathcal{T}$ be the set of threats and consecutive

Table 5.3: Composed Features.

Composed Feature	Related Atomic Features
Response type	{code,token,client_credentials,password,hybrid}
Request	{request,request_uri,query}
Response mode	{form_post,fragment,query}
Code challenge method	{plain,SHA256}
Client authentication	{mTLS,client_secret_jwt,private_key_jwt,client_secret_basic,client_secret_post}
Redirect uri	{full redirect_uri,pattern redirect_uri}
Identity proofing	{vot,acr_values}
Prompt	{login,select_account,consent}
Subject type	{public,pairwise}

threats, listed in columns 3 and 5 of Table 5.1. Let $\mathcal{G}$ be the set of goals and consecutive goals, listed in columns 4 and 6 of Table 5.1. We thus define the following mappings and relations:

- let p be a mapping from $\mathcal{AF}$ to the set $\{1,\ldots,5\}$ of protection levels. The definition of this mapping can be obtained by considering the features in column 2 and the corresponding protection level in column 7 of Table 5.1.
- let the likelihood mapping ℓ be a mapping from $\mathcal{CF}^*$ to the set $\{1,\ldots,5\}$ of likelihood levels. The definition of this mapping can be obtained by considering the (sets of) atomic features in column 2 and the corresponding likelihood level in column 8 of Table 5.1.
- let $CF2T \subseteq \mathcal{CF}^* \times \mathcal{T}$ be a relation between each composed feature $cf \in \mathcal{CF}^*$ and its related threat. The pairs in this relation can be defined by reading the elements reported in columns 2 and 3 of Table 5.1. In the rest of this chapter, we also use the notation $T(cf)$ to indicate the set of threats related to a composed feature cf.
- let $T2G \subseteq \mathcal{T} \times \mathcal{G}$ be a relation between each threat and the goal compromised by the threat itself. The pairs in this relation can be defined by reading the elements reported both in columns 3 and 4, and 5 and 6 of Table 5.1. Indeed, the relation between a threat and the goal is independent of the fact that the threat is consecutive or not of another threat. We also use the notation $g(t)$ to indicate the goal related to a threat $t \in \mathcal{T}$.
- let $T2CT \subseteq \mathcal{T} \times \mathcal{T}$ be a relation between a threat and its consecutive threat. The pairs in this relation can be defined by reading the elements reported in columns 3 and 5 of Table 5.1. We also use the notation $CT(t)$ to indicate the set of threats consecutive to the threat t, and $CT(T) \triangleq \bigcup_{t \in T}(CT(t))$.

The problem P_{risk} consists in finding the *idmp* and mapping $implement_{cl}$ such that

$$min_{idmp,implement_{cl}} \mathcal{R}(p, \ell, support_{idmp}, implement_{cl}, CF2T, T2G, T2CT) \tag{5.1}$$

subject to $idmp \in$ IdMPs and $cl \in ClAdm$, where IdMPs is a set of IdMPs that support certain features and $ClAdm$ is the set of admissible mappings for a given client.

The set IdMPs and the admissible mappings $ClAdm$ come from external considerations performed by the controller of the client. For instance, IdMPs can be affected by the costs of their services or which kind of features they support. Similarly, the controller can consider that some features—among the ones constrained (as defined above) by the selected *idmp*—will take longer to implement or charge more in terms of costs. Thus, the controller can further constrain the admissible $implement_{cl}$ mappings accordingly.

$\mathcal{R}$ is thus an operator that returns the risk level by solving the P_{risk} problem as illustrated in the formula (5.1). While p,l, $CF2T$, $T2G$, and $T2CT$ values are constant (obtained from Table 5.1),

$support_{idmp}$ and $implement_{cl}$ can change. Note that, we provide a database of known IdMPs in Table 5.4. We specify the $support_{idmp}$ Boolean mappings for these IdMPs by considering *af*s that their values can be identified by using the IdMP's documents. In order to collect the information w.r.t. the missing *af*s information, one possible solution is to ask app developers to reply a dynamically generated Yes/No questionnaire.

Therefore, the problem is decidable because it can be expressed as a combinatorial optimization problem with many finite possibilities depending on the number of considered IdMPs and on the cardinality of the considered atomic features in $\mathcal{AF}$: all the possibilities can be enumerated, the corresponding risk evaluated and the minimum value selected. This problem can be fed into an automated risk modeling and evaluating tool to provide an output that enables app developers to minimize the risk by making an informed decision for their OAuth/OIDC solutions. A representative example of such kind of tool can be illustrated by RiskML [SMS14]. RiskML allows app developers to perform a risk assessment taking into account the risk propagation and also supports a trade-off analysis, allowing app developers to consider the risk associated with each implementation choice.

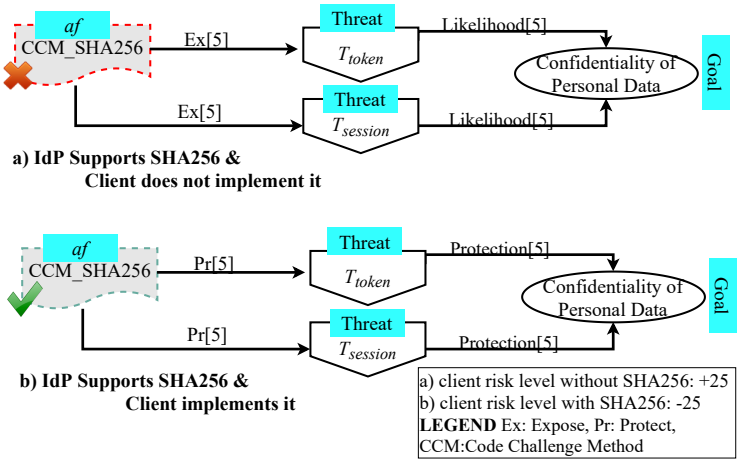
To illustrate the advantages of performing the OAuth/OIDC risk assessment and how it can help app developers to make an informed decision, let us consider a simplified example for the *cf code challenge method* represented in Figure 5.1. Note that, we assume that the IdMP supports *af* `SHA256` of *cf code challenge method*. Figure 5.1.a illustrates the scenario that the app developer does not implement the *af* `SHA256` that leads to T_{token} and $T_{session}$ threats with the likelihood level

Table 5.4: List of Known IdMPs and their Supported Features.

Atomic Feature	PING	IBM	Google	Facebook	Yahoo	Amazon	Linkedin	Microsoft	DropBox	OKTA	Auth0	Box	Hidrive	BeeMinder
code	T	T	T	T	T	T	T	T	T	T	T	T	T*	F
token	T	T	T	T*	T	T	F	T	T	T	T	F	F	T
client_credentials	T	T	F	F	F	F	T	F	F	T	T	F	F	F
password	T	T	F	F	F	F	F	F	F	T	T	F	F	F
hybrid	T	T	F	F	F	F	F	T	F	T	T	F	F	F
mTLS	T	T	F	F	F	F	F	F	F	F	F	F	F	NA
client_secret_basic	T	T	T	T	T	T	T	T	T	T	T	T	T	NA
client_secret_post	T	T	T	T	T	T	T	T	T	T	T	T	T	NA
client_secret_jwt	F	F	F	F	F	F	F	F	F	T	F	F	F	NA
private_key_jwt	T	T	F	F	F	F	F	T	F	T	F	F	F	NA
plain	T_opt	T_opt	T_opt	F	F	T	F	T	T	F	T_opt	F	F	NA
S256	T_opt	T_opt	T_opt	F	F	T	F	T	T	T	T_opt	F	F	NA
request	T	T	F	F	F	F	F	T	F	T	F	F	F	F
request_uri	T	T	F	F	F	F	F	F	F	F	F	F	F	F
query	T	T	T	T	T	T	T	T	T	T	T	T	T	T
claim	F	T	F	F	F	F	F	F	F	F	F	F	F	F
scope	T	T	T	T	T	T	T	T	T	T	T	T	T	T
purpose	T	F	F	F	F	F	F	F	F	T	F	F	F	F
verified_claims	T	F	F	F	F	F	F	F	F	T	F	F	F	F
acr_values	T	T	F	F	F	F	F	T	F	T	T	F	F	F
vot	F	F	F	F	F	F	F	F	F	T	F	F	F	F
public	T	T	T	NA	T	T	NA	F	NA	T	T	NA	NA	NA
pairwise	T	F	F	NA	F	F	NA	T	NA	F	F	NA	NA	NA

LEGEND. T_opt:supported but not enforced, T*:not follow the standard definition, NA: not applicable

Figure 5.1: An Excerpt Risk Model, Related to Feature *Code Challenge Method*.



of 5 and consequently compromise the goal *confidentiality of the personal data* (as reported in Table 5.1). Given that, the client risk level for this situation is equal to 25 (5 * 5). While Figure 5.1.b represents a scenario where the app developer had implemented the *af* `SHA256` that protects the client against the T_{token} and $T_{session}$ threats with the protection level of 5 and consequently protect the goal *confidentiality of the personal data*. In this way, app developers can observe risk posed by overlooking the implementation of the *af* `SHA256` in comparison with the scenario that s/he would implement the aforementioned *af*. Thus, it enables app developers to learn how to decrease the risk level by changing their implementation choices.

Chapter 6

mIDAssistant Plugin

In this chapter, we present mIDAssistant: android Studio plugin for assisting native app developers with automatic and secure integration of OAuth/OIDC solutions within their native apps.[1]

As it is shown in Chapter 4, unfortunately, many native apps under analysis still do not follow the security BCPs as native app developers are using known anti-patterns, such as the *Implicit* grant flow. Although IdMPs make an effort to help native app developers by providing their official SDKs, our analysis proves that some official SDKs are not following the OAuth/OIDC security BCPs. Thus, the usage of official SDKs do not necessarily mean that the integrated solution by native app developers is secure. These problems can be mitigated by adopting an approach that automates the enforcement of the security BCPs for OAuth/OIDC solutions within native apps. Thus, it motivates us to provide a wizard-based approach and implements it within mIDAssistant plugin aiming at supporting native app developers with secure integration of OAuth/OIDC solutions within their native apps. Note that we do not aim to provide a solution to improve the usability as our proposed method is more focused on security by design. In particular, mIDAssistant plugin provides a way to "enforce" the usage of security BCPs for native apps by using the AppAuth SDK.

Section 6.1 explains our wizard-based approach by detailing its two main processes, namely DB population and Wizard questions. In Section 6.2, we detail these two processes for *fully compliant* IdMPs, and Section 6.3 summarizes the required changes for *not fully compliant* IdMPs. Finally, the plugin architecture together with a security analysis results on the integrated code by the mIDAssistant plugin is given in Section 6.4.

[1] This chapter is based on our published work in [SCRS19].

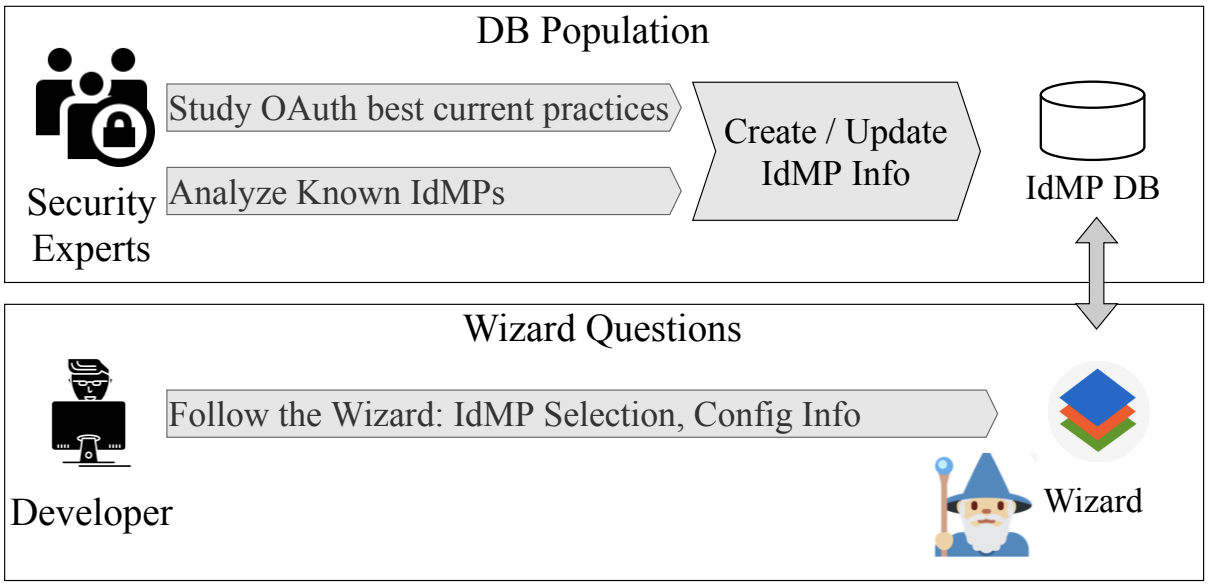


Figure 6.1: Approach.

6.1 Wizard-based Approach

As depicted in Figure 6.1, our wizard-based approach encompasses two processes, namely DB population, and Wizard questions. In the former, security experts (e.g., members of the OAuth WG) populate the database of supported IdMPs. In the latter, possibly security inexperienced app developers—taking advantage of the information given by security experts about supported IdMPs—select one of them and follow the instructions of the wizard, by providing the missing information concerning their own apps. The following sections detail these two processes for *fully compliant* IdMPs, namely the IdMPs that follow all recommended BCPs for native apps (cf. Table 3.2) and consequently require no changes within the AppAuth SDK, and *not fully compliant* IdMPs, namely the IdMPs that do not follow at least one of the recommended BCPs for native apps, but they can be still used in a secure manner. If this is the case, some changes in the code of AppAuth SDK are required, like omitting *PKCE* and adding the `client_secret` as they are not compatible with the default configuration of AppAuth SDK.

6.2 Support fully compliant IdMP

In our analysis (cf. Table 4.1), Google, Microsoft v2, OKTA, Ping, IBM, Amazon and Auth0 follow all recommended BCPs for SSO and/or AD for native apps, thus we consider them as *fully compliant* IdMPs.

6.2.1 Security Experts: DB population

Let us consider the information that security experts are required to provide for each new provider *IDMP*. At first, the security experts must evaluate whether the IdMP is *fully compliant* with AppAuth, namely if it follows what reported in Section 3.3.1. If this is the case, they should specify which are the scenarios offered by *IDMP*, that is, SSO (*IDMP*.`Scenario=SSO`), Access Delegation (*IDMP*.`Scenario=AD`), or both (*IDMP*.`Scenario=both`). Another information is related to the supported redirection mechanisms during *C* app registration: IdMP supports only HTTPS (*IDMP*.`Redirect=HTTPS`), only Custom URI scheme (*IDMP*.`Redirect=CustomURI`), or both (*IDMP*.`Redirect=both`). In addition, if discovery URL is supported, they should specify the discovery URL (*IDMP*.`Discovery=`*Value*), otherwise they should provide *authorizationInfo* (*IDMP*.`AuthnURL=`*Value*) and *tokenInfo* (*IDMP*.`TokenURL=`*Value*) Endpoints. In some cases, these endpoints are generated starting from a domain that is assigned to the developer. In these cases, the security expert must specify this setting *IDMP*.`DevDomain=yes`. If *IDMP*.`Scenario=both` or *IDMP*.`Scenario=SSO` the *userInfo* Endpoint (*IDMP*.`UserInfoURL=`*Value*) must be provided as well.

While some of the aforementioned information (e.g., the endpoints) is used to properly configure AppAuth, the others (e.g., supported scenario and redirection scheme) are used by the wizard to customize the questions for the app developer, as detailed in the following subsection.

6.2.2 Developers: Wizard Questions

In Listing 8.1, we clarify which are the questions asked to app developers through the wizard (in *italic*) and how they are customized according to the information about IdMPs. While the complete code integration process is detailed in Section 6.4, the Listing 8.1 is mainly focused on the interaction of app developers with the wizard, which is a preliminary step to instantiate the AppAuth customized code and automatically integrate it within an app.

Listing 6.1: Wizard for fully compliant IdMPs.

```
// Scenario
Choose whether you want to add in your app a button for either SSO Login, or AD
scenario := user_pick({SSO Login, AD})
If scenario = SSO Login then
  IDMPS := {IDMP | IDMP ∈ DB ∧ IDMP.Scenario ∈ {both, SSO}}
If scenario = AD then
  IDMPS := {IDMP | IDMP ∈ DB ∧ IDMP.Scenario ∈ {both, AD}}

// Selection
Please, select IDMP among the list of supported IdM Providers
```

IDMP := user_pick(*IDMPS*)
If *IDMP*.`Discovery` is set to a value
  then Create(appauth_config, *IDMP*.`Discovery`)
  else Create(appauth_config, *IDMP*.`AuthnURL`, *IDMP*.`TokenURL`)
If scenario = SSO Login then
  Update(appauth_config, *IDMP*.`UserInfoURL`)
If scenario = AD then
  Please, enter the resource endpoint
  resourceURL := user_insert_value
  Update(appauth_config, resourceURL)

// Configuration
Enter Configuration Info: Scopes (optional), ClientID, the Name of the Button you want to add
Scopes := user_insert_value
ClientID := user_insert_value
ButtonName := user_insert_value
Update(appauth_config, Scopes, ClientID, ButtonName)
If *IDMP*.`DevDomain=yes` then
  Please, enter your developer domain
  devURL := user_insert_value
  Update(appauth_config, devURL)

// Redirection
If *IDMP*.`Redirect=both` then
  Choose the preferred Redirection Method: either HTTPS, or CustomURI
  redirect_method := user_pick({HTTPS, CustomURI})
If *IDMP*.`Redirect=HTTPS` or redirect_method = `HTTPS` then
  Enter your valid domain URL: scheme, host, and path
  scheme := user_insert_value
  host := user_insert_value
  path := user_insert_value
  Update(appauth_config, scheme, host, path)
If *IDMP*.`Redirect=CustomURI` or redirect_method = `CustomURI` then
  Enter the Custom URI of your app
  uri := user_insert_value
  Update(appauth_config, uri)

According to the choices of the app developer, either OIDC or OAuth is used, and the wizard enforces the proper implementation by automatically adding the correct code within the developer app. This procedure is detailed furtherly in Section 6.4. It is important to remark that the wizard helps developers to use the proper flow according to the selected scenario to avoid, for instance, the common mistake of using OAuth, instead of OIDC, for authentication purposes.

6.3 Support not fully compliant IdMPs

In our analysis (cf. Table 4.1), Yahoo, DropBox, and Box are *not fully compliant IdMPs*. Yahoo is violating BCP_{flow} and BCP_{PKCE}, Box BCP_{UA}, BCP_{PKCE}, and BCP_{flow}, and DropBox only BCP_{flow}. Our approach provides a way to support the aforementioned IdMPs in a secure manner by identifying the following changes:

- to omit the sending of the *PKCE* query parameters whenever the IdMP's server does not support the *PKCE* protocol, but at the same time it allows the HTTPS scheme and provides a CSRF protection mechanism, such as `state` or `nonce`. As our approach enforces the HTTPS scheme together with `state` and `nonce` query parameters in the authorization request, it provides a way to distinguish the authentic *C* app (through HTTPS scheme), and avoids possible CSRF attacks (usage of `state` and/or `nonce`). This solves the problem with BCP_{PKCE} of Yahoo, and Box. Indeed, the proper usage of HTTPS scheme together with `state` and/or `nonce` query parameter can provide the same level of security as BCP_{PKCE}.
- to enforce the sending of the `client_secret` whenever its value is not actually used as a secret and it is necessary due to the legacy reason. Although code hardening and obfuscation techniques are efficient against static source code analysis, they are not helpful against dynamic code analysis. Given that, developers are suggested to use Dynamic Client Registration (DCR) method in [JBMH15] (whenever it is applicable), to obtain per app instance client credentials.
- to enforce the usage of in-app browser tabs as *UA*, solving the problem of BCP_{UA} (e.g., in Box).

It is worth mentioning that the sending of the `client_secret` is not supported in the default implementation of our wizard, as it is based on BCPs for native apps, which recommend not to use the `client_secret`. The only exceptions that our approach permits are for solutions where the `client_secret` is not used to authenticate the client, but it is needed for legacy reasons. Thus, a security expert should verify that the `client_secret` used for a native app is, at least, different from a secret released for web applications.[2] In addition, if the IdMP supports the DCR method to register a new *C* app with an IdMP, app developers must use this technique to obtain a different `client_secret` for each app installation that limits the impact of the `client_secret` leakage into the specific instance of the *C* app. In the case that IdMPs rely on the "secrecy" of the `client_secret`, as future work we plan to: require the Dynamic Client Registration (DCR), in such a way to have a different `client_secret` for each app installation; or perform the code exchange step through the backend of the app (if it is present) that can securely store the `client_secret`.

[2]Based on our IdMPs analysis, it seems some IdMPs are using the same client credentials for both web and native clients, and we are in process of validating our finding with related IdMPs.

Listing 6.2: Wizard for not fully compliant IdMPs.

```
// Redirection                                                          1
If IDMP.PKCE=no then                                                    2
  Enter your valid domain URL: scheme, host, and path                   3
  scheme := user_insert_value                                           4
  host := user_insert_value                                             5
  path := user_insert_value                                             6
  Update(appauth_config, scheme, host, path)                            7
else the same behaviour of Listing 8.1, lines 34-46                     8
                                                                        9
// Client Secret                                                        10
If IDMP.Secret=yes then                                                 11
  Enter the App Client Secret                                           12
  ClientSecret := user_insert_value                                     13
  Update(appauth_config, ClientSecret)                                  14
```

6.3.1 Security Experts: DB population

As a consequence, in case an IdMP is *not fully compliant* with AppAuth, the security experts must evaluate whether it is due to:

- the lack of *PKCE* and the IdMP supports HTTPS scheme, and if this is the case they should specify *IDMP*.`PKCE=no`, and/or
- the need to include a `client_secret` for legacy reasons. The security expert should check that it is not considered by *IDMP* as a real "secret". If this is the case they should specify *IDMP*.`Secret=yes`.

6.3.2 Developers: Wizard Questions

Concerning the changes in the wizard, while the Scenario, Selection and Configuration parts are the same of Listing 8.1, in Listing 6.2, we clarify how the other questions change in case an *IDMP* is *not fully compliant* with BCPs. In detail: in the Redirection part, if the IdMP does not support the *PKCE* protocol (line 2) then we enforce an HTTPS scheme, by asking to the user the needed information and updating the configuration (lines 3-7). In the Client Secret part, if the IdMP requires a `client_secret` (line 11), then its value is asked to the developer and the configuration is updated accordingly (lines 12-14).

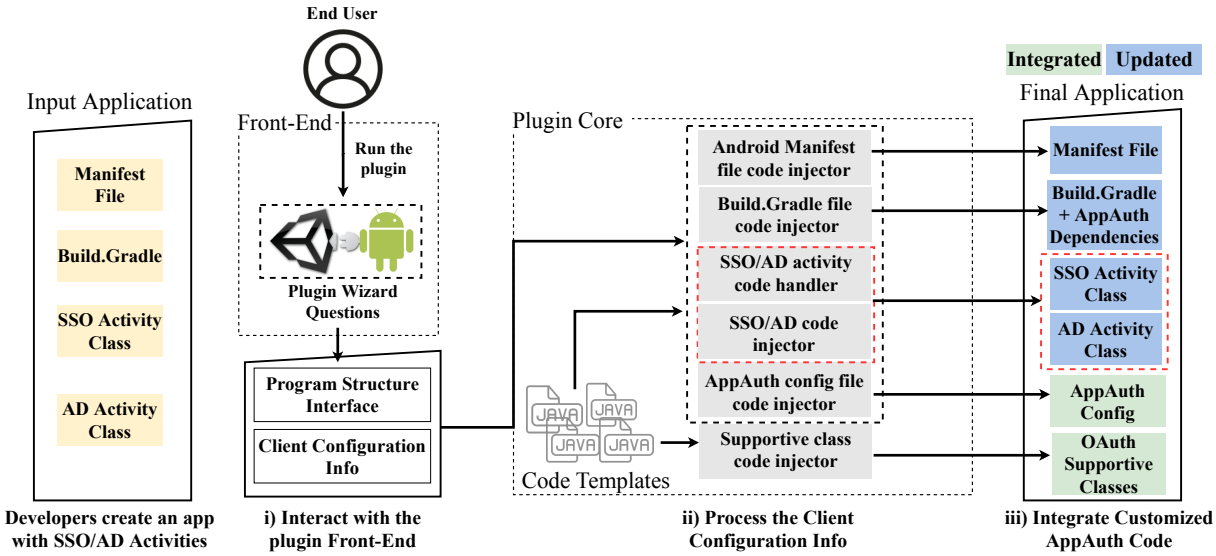


Figure 6.2: mIDAssistant Plugin Workflow.

6.4 Implementation and Experimental Analysis

We implemented our wizard-based approach in mIDAssistant, an open-source Android Studio plugin[3] developed using the Intellij Idea environment (almost 4500 lines of code), which can be integrated in the development environment with few clicks. The current version of the plugin is able to communicate with Auth0, OKTA, Google, Ping, IBM and Microsoft as they are *fully compliant* with AppAuth, and with Yahoo, DropBox, and Box after the changes explained in Section 6.3. Compared to the preliminary version of mIDAssistant presented in [SCRS19], we added the support for Ping and IBM and we updated the IdMP endpoints to the last available versions.

In Section 6.4.1, we explain the workflow of the mIDAssistant plugin and the detail code integration process. Section 6.4.2 discusses the security of the code integrated within a demo app supporting the login with OKTA and Google by using the mIDAssistant plugin. Furthermore, for interested readers we provide a demo to show how the mIDAssistant plugin can be used to effortlessly and securely integrate the code to add the aforementioned functionalities[4].

[3] https://github.com/stfbk/mIDAssistant

[4] https://github.com/stfbk/mIDAssistant#usage

6.4.1 mIDAssistant Workflow

Assuming that an app developer creates a *C* app with SSO/AD activity classes beforehand or s/he wants to add SSO/AD functionality to an already developed native app, as shown in Figure 6.2 the mIDAssistant plugin workflow contains three main steps, namely: (i) interacting with the front-end, (ii) processing the obtained information from the front-end in the core, and (iii) integrating the customized AppAuth code within the developer's app. In Step (i), app developers run the mIDAssistant plugin to interact with the plugin front-end to compile wizard questions as explained in Listing 8.1. Therefore, this step provides the client configuration info and the Programming Structure Interface (PSI) as an output. While the PSI provides a way to access the source code of *C* app as a hierarchy of elements that is used by code injectors and code handler, the client configuration info provides the necessary information to configure the *C* app that is needed for the code integration process within the plugin core. After that, in Step (ii), the plugin utilizes the code templates besides the input from the previous step (client configuration info and PSI) to initialize the code injector/handler within the plugin core. It is worth mentioning that the code templates are mostly obtained from the AppAuth repository.[5] In the following, we explain the functionality of each code injector/handler component within the plugin core. In case developers choose the HTTPS scheme within the GUI, the *Android Manifest file code injector* extracts the HTTPS value from the client configuration info to initialize the customized code to handle the HTTPS redirection. The *Build.Gradle file code injector* initializes the customized code to add the AppAuth SDK dependencies. In addition, it handles the Custom URI redirection by obtaining its relevant value from the client configuration info in case of Custom URI selection by developers. The *AppAuth config file code injector* initializes a JSON structure that obtains its values from the relevant parameters in the client configuration info, which is necessary to enable app communication with specific IdMPs. The *SSO/AD code injector* and *SSO/AD activity code handler* initialize the customized code to add main SSO/AD functionalities within SSO/AD activity classes. They accomplish the aforementioned task by obtaining the relevant values and code from the client configuration info and code templates, respectively. While *SSO/AD activity code handler* adds the initialization code for the SSO/AD button within the `OnCreate` method in the SSO/AD activity classes, the *SSO/AD code injector* provides the main functionalities in the above mentioned classes for the SSO/AD button. In contrast to previous code injectors, the *Supportive class code injector* initializes the customized code only from code templates (Figure 6.2) which provides necessary OAuth/OIDC functionalities, such as: read/validate the AppAuth Config file. Finally, in Step (iii), the code injector/handler incorporates the customized code into the app to update Build.gradle, Android Manifest, and SSO/AD activity class files. It also integrates newly created AppAuth Config and OAuth supportive classes files.

The plugin integrates necessary files to enable app communication with specific IdMPs and leverages the AppAuth SDK as the main core to ensure the enforcement of BCPs for native apps. Therefore, our solution helps developers eliminate the maze of the OAuth/OIDC related docu-

[5] `https://github.com/openid/AppAuth-Android`

ments and the corresponding security considerations to avoid wrong implementation choices. In addition, as the plugin takes care of the SSO/AD integration, developers can focus on other tasks which are out of the scope of the plugin, such as code obfuscation, certificate pinning.

The mIDAssistant plugin can be used to add SSO/AD functionalities within an already developed app without raising conflicts and/or compatibility issues. Indeed, native app developers only needs to write a few lines of code to properly handle the newly added SSO/AD functionalities within their app.[6]

It is worth mentioning that the design and implementation of the plugin was not an easy straightforward task as we faced the following challenges:

- To provide the relevant implementation questions within the plugin GUI, we have extracted all the necessary information related to the OAuth/OIDC implementation from the OAuth/OIDC specifications. For example, the OAuth specification makes the usage of the 'Scope" parameter optional, thus if native app developers leave the relevant question empty in the GUI, the plugin will not pop-up an error concerning "Scope" information is missing, while this is not the case for the OIDC.
- To implement the *SSO/AD activity code handler*, we had to study and practice with PSI elements to access the specific part of the source code programmatically and integrate the plugin code;
- To implement the *AppAuth Config file code injector* and *Android Manifest file code injector*, in addition to the PSI elements to access the proper file, we had to deal with how to traverse programmatically within AndroidManifest and the automatically-generated AppAuth Config file to find the proper place to integrate the relevant plugin code.

As a final remark, note that the mIDAssistant plugin is still in an early stage release and contains the following limitations: (i) native app developers need to create a specific folder under their project directory, which is used by the mIDAssistant plugin to integrate the "AppAuth Config" file; and (ii) the integrated code for incorporating multiple IdMPs still needs some improvement.

6.4.2 Security Analysis

In this section, we perform a security analysis of the code that is integrated in an app using the mIDAssistant plugin to invoke the OAuth/OIDC functionalities from the AppAuth SDK and enables *C* app communication with integrated IdMPs. We have personally developed (150 lines

[6]The current implementation of plugin cannot assist native app developers to turn their non-compliant SSO/AD solutions into compliant one by using our plugin.

of code) to support multiple IdMPs integration within apps. Therefore, it is a very small portion of the whole integrated code by the plugin in comparison with the reused code templates (600 lines of code) and AppAuth SDK (9400 lines of code). Thus, the security of our approach concerning the SSO/AD implementations is mainly depend on the AppAuth SDK and then the integrated code by the plugin. In our security analysis procedure, we consider the AppAuth SDK source code secure based on the following assumptions: AppAuth SDK is an open-source project endorsed by the OpenID Connect Foundation since 2016; and it is based on the code developed by Google Identity Platform team and reviewed by the Google's internal security team. Given that, we verify only the security of the integrated code within the developer's app by using both open-source and commercial source code analysis tools. Furthermore, to increase the trust in the correctness of the plugin, we make its code open-source as it provides a way to perform a code-review process by the experts in this field. In the following, first we elaborate on the selection of open-source and commercial dynamic analysis tools and then summarize our findings based on the security report of the selected tools.

6.4.2.1 Selected Static/Dynamic Analysis Tools

We conduct a comprehensive study on available static/dynamic analysis tools in the wild to perform a security analysis on the integrated plugin code. While for the static analysis tools, we used the result of our analysis report in Section 4.3.3. In the case of dynamic analysis tools, we analyzed 16 dynamic source-code analysis tools that are available online. 7 out of 16 are open-source tools, while the rest are commercial tools. We provide more detailed information for the considered tools in Appendix C for interested readers.

Furthermore, we checked the tools for their capabilities, such as web view detection, dynamic code loading, HTTP traffic analysis, etc. We selected two open-source (Super, MobSF) and one commercial (Approver) tools to perform the security analysis on the integrated code based on the obtained results. Below, we detailed our findings based on the security reports provided by the selected tools.

Open-Source Tools. In this phase, we perform a security analysis on the integrated mIDAssistant tool code by utilizing two open-source tools, namely: Super and MobSF [Sup18, Ope18]. The result of our analysis is summarized as follows:

1. Weak Cryptography Algorithm: Super source code analysis tool detects the usage of a weak cryptography algorithm. Weak cryptography algorithm usage can allow an attacker to break the ciphered communications and access plain text content.
2. Security Information Disclosure: MobSF source code analysis tool detects some strings that are classified as secret keys.

With a further analysis, we discover that both results are not security issues in our case. Indeed,

Super raises a false alarm about the weak cryptography algorithms usage, as the detected weak algorithm is used to create a hash of the configuration file. The SDK uses the hash to find the change in the configuration file that leads to authorization state clearance. The second item is not a security issue. Indeed, the result was FP as it marks all the plain strings with a certain length as possible secret keys.

Commercial Tool. Approver provides a software as a service solution for the in-depth, fully automated security analysis and risk assessment of Android apps [VAMC18]. The security report of the tool contains the following issues:

1. SSL Implementation (Verify Host Name in custom class): The app accepts all Common Names (CN) that allows the attacker to perform MiTM attacks.
2. SSL Certificate Verification Checking: The app does not check the validation of SSL certificate. Indeed, it allows a self-signed, expired, or mismatched CN certificate for SSL connection, making it vulnerable to MiTM attacks.
3. Code Obfuscation: the app seems not using sufficient code obfuscation.
4. Certificate pinning: the host contact URLs under HTTPS connection, but without certificate pinning.
5. Undesired access to private data: the app should prevent undesired access to private data and preferences.

The first two findings are originating because the plugin's integrated code provides a connection builder class for testing purposes that ignores Host Names and SSL Certificates. Therefore, we can avoid them by using the AppAuth SDK instance directly. As the main goal of our plugin is to help app developers with the integration of multiple IdMPs within their app, the reported findings through 3 to 5 are considered out of scope for our plugin solution. Therefore, we delegate the tasks of performing code obfuscation, encrypting the information in preferences and certificate pinning to app developers. We repeat the source code security analysis after we substitute the connection builder class for testing with the main connection class in the AppAuth SDK and apply the certificate pinning to confirm our claims concerning the suggested remediation. The final security report confirms that the app does not contain any issues related to the SSL implementation and certificate pinning.

Chapter 7

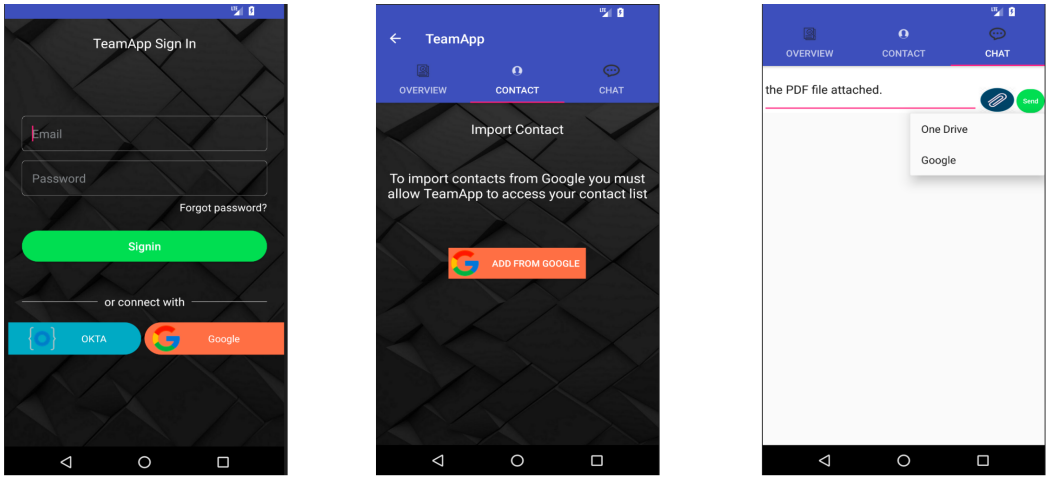
mIDAssistant Applications on the Real World Scenarios

In this chapter, to show the effectiveness of our mIDAssistant plugin that we have introduced in Chapter 6, we will apply it within real-world scenarios that are dealing with secure integration of multiple IdMPs. To this aim, we first define the problems that native app developers may be encountered in the multiple IdMPs scenario in Section 7.1. Then, we explain how our mIDAssistant plugin can help native app developers to address the identified problems during implementation process for the identified scenarios, namely: Teamapp (Section 7.1.1), Pull Printing (Section 7.1.2), and API Assistant (EIT activity - Section 7.1.3) scenarios.

7.1 Multiple IdMPs

As mentioned in Section 1.1, app developers are still failed in secure integration of OAuth/OIDC solutions. However, the situation can become more complex for scenarios that require integration of multiple IdMPs. We conjecture that the following conditions have led to this situation:

- available information about BCPs for native apps is sparse and it is not easily understandable and consequently actionable by the average of app developers.
- lack of BCPs adoption by OAuth/OIDC providers that leads to a twofold negative consequence: difficulty in integration of AppAuth SDK within their apps and additional burden on app developers who need to understand and implement a variety of ad hoc security measures for different IdMPs that would lead to insecure implementations.
- lack of guidelines for the secure integration of multiple IdMPs within a single app—an increasingly popular use case scenario in modern apps.



a) Login Activity b) Contact Activity c) Chat Activity

Figure 7.1: TeamApp Screen Shots.

In the following, we explain three different scenarios that app developers need to deal with the integration of multiple IdMPs and summarize how our approach can help app developers during the implementation process.

7.1.1 TeamApp Scenario

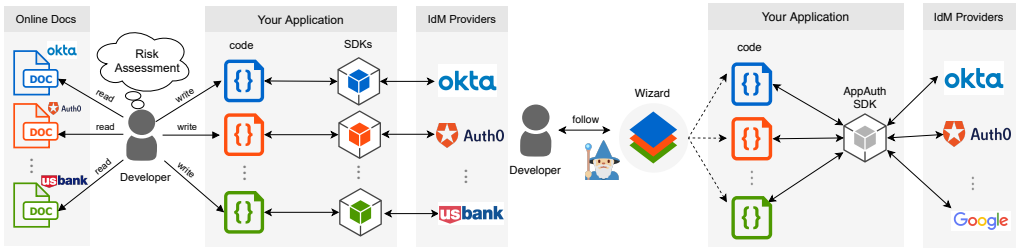
Let us consider a scenario that *university A* is collaborating with *research center B* on a joint project. To accelerate their collaboration and communication between team members, they decide to develop the *TeamApp*.

To access heterogeneous services on *TeamApp*, the user should login into *TeamApp* by using either traditional login methods or SSO solutions (e.g., OKTA or Google as shown in Figure 7.1.a.). After authentication, a *User* can (i) create a new team by choosing to import her Google Contacts by navigating to the Contacts tab within the app (Figure 7.1.b) and clicking on the button *add from Google*, and (ii) share documents with the team using the Chat tab and clicking on the *attachment* button to import the document from external providers (e.g., OneDrive and Google as shown in Figure 7.1.c).

The effort required to implement *TeamApp* is substantial. As shown in Figure 7.2a, a developer should:

P1. refer to the documentation of each IdMP;

P2. download the different SDKs and understand how to interact with them to implement the IdM solution;



(a) Without our approach. (b) With our approach.

Figure 7.2: The Comparison between Current and Proposed Approach.

P3. write different pieces of code to integrate the SDKs.

Furthermore, in such scenarios where multiple IdMPs must be integrated, AppAuth cannot be easily adopted because its documentation seems to support the interaction with a single IdMP. Therefore, app developers have two options to deal with several IdMPs at the same time: either reuse the same classes and configuration files of AppAuth by renaming them in order to avoid conflicts or implement ad hoc methods from scratch. Both options are not only time-consuming but also error-prone, especially for inexperienced app developers.

To overcome these difficulties, app developers can use our approach (shown in Figure 7.2b) that is implemented within mIDAssistant plugin and automates most of the process by:

S1. avoiding the need for reading online documentations (solving P1): it provides a built-in list of IdMPs with their related information (e.g., endpoints);

S2. avoiding the need for downloading several SDKs (solving P2): it is based only on the AppAuth SDK that is (effortlessly) integrated once and for all;

S3. automatically integrating the code to incorporate AppAuth within the app developers and enable the communication with multiple IdMPs (solving P3).

In this way, our wizard-based approach allows app developers to integrate multiple IdMPs effortlessly and in a secure manner. In particular, app developers simply interact with the mIDAssistant GUI to select their preferred IdMP and to provide the mandatory information (See Sections 6.2 and 6.3). After that, the plugin automatically integrates the secure customized code within the app developer's app.

7.1.2 Pull Printing Scenario

Pull printing–also called "follow-me printing"– enables users to submit their print job within their local machine to a local or remote server and release them later according to specific conditions. In particular, the user needs to authenticate at printer either using hardware-based (e.g., embedded software or external print release appliances) or software-based solutions (e.g., print orchestrator), and after the authentication, the user may select the print jobs they wish to be released. Organizations typically adopt pull printing to leverage the following benefits:

- pull printing adds an additional layer of security by providing a way to guarantee that users control the print job. This means that they can execute the print job exactly where and when they need it to make sure it ends up in the right hands. Thus, it mitigates the risk related to the abandoned confidential or important documents in the printing tray for others to view or even take;
- pull printing reduces the printing waste as it simply sends the print job to the print queue and releases it where and when it is convenient for users. Therefore, it avoids situations that users may forget their printed document, and when they remember, the document has either been recycled and has to be re-printed.
- pull printing improves user productivity by enabling users to send their print job to print queue and pick it up in bulk later. Also, if one printer is not functioning, they can always choose an alternative printer to complete the process;
- pull printing is complied with the recent EU data protection regulations, such as article 32 of the EU GDPR [GDP16].

It is worth mentioning that there is a positive attitude within the organization concerning adopting the pull printing solutions, as stated in a report by QuoCirca [Quo19]. The reason for the increase, as mentioned earlier, can be due to the security impact of the print-related security incident, which led to data loss and damage the enterprise business, especially considering the fines prescribed by current regulations [GDP19], or its reputation: therefore, loss of clients and revenue.

The FBK pull printing service [LMSR21] extends a Print Orchestrator's capability with a multi-platform, printer agnostic system that authenticates employees on an app and requires a second-factor authentication by using the eID card (Italian digital Identity card) for the sensitive documents. The use of eID cards enables enterprises to avoid issuing new smart cards or customize (if even possible) employee badges. In the following, I will explain how app developers are empowered by mIDAssistant plugin to implement the pull printing functionality within a print releaser app developed in our research unit.

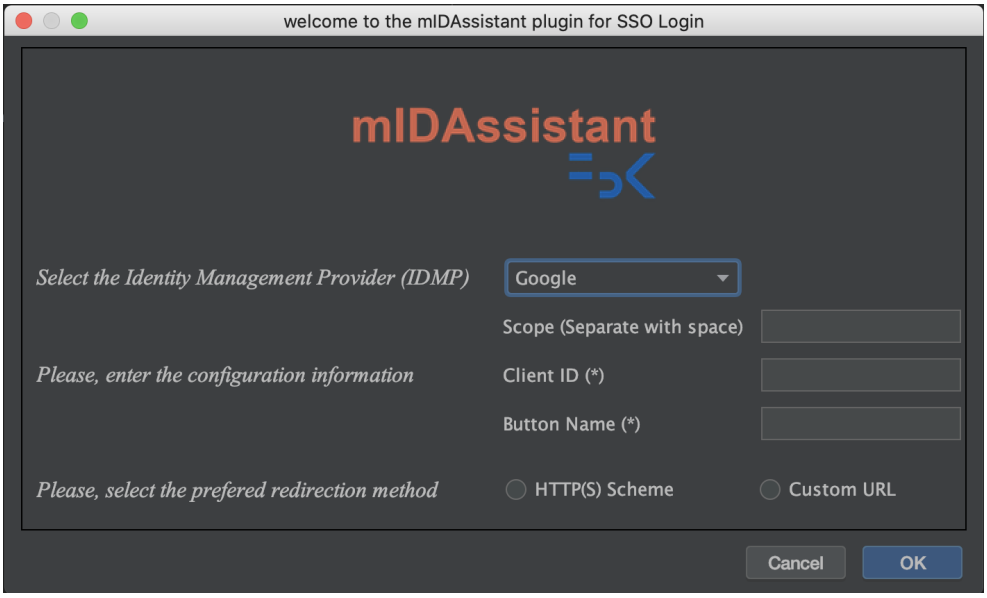


Figure 7.3: mIDAssistant GUI.

7.1.2.1 Print Releaser App development

One of the main challenges in the print releaser app development is related to employee authentication. The authentication servers to authenticate the employee and validate the eID cards can be managed on-premise or be external services federated with the enterprise. In the FBK deployment of the print releaser app[1], the Google Suite platform is used to handle the employee accounts. In addition, the *Authorization and Authentication Control (AAC)* module developed by Smart Community is used as an OIDC server, which provides a sub-module for the second-factor authentication via the Italian eID card.

To provide the SSO Login with Google and AD towards Google cloud print and AAC functionalities, app developers need to implement the OIDC *Authorization Code* flow with *PKCE* within their apps. To do so, app developers use our plugin–mIDAssistant– to interact with its GUI, which enables them to provide the necessary information (`client_id`, `scope`, `redirect_uri` and `client_secret` (if needed)) to integrate the secure code within their app. Figure 7.3 represents the plugin GUI together with the necessary information that app developer must provide. Therefore, the plugin completely handles the authentication/authorization part of the pull printing service and frees the app developer from understanding the maze of documents and their related security implications. Indeed, as the plugin enforces the latest BCPs, app developer can focus on other tasks not covered within our plugin.

[1] `https://github.com/stfbk/pullprinting`

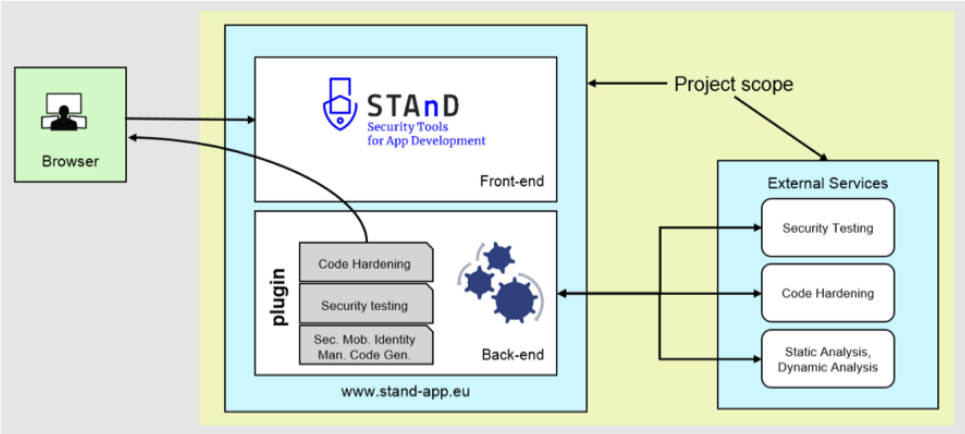


Figure 7.4: The High-Level Architecture of STAnD Project.

7.1.3 API Assistant

In this section, I summarize the integration of our approach within API Assistant Innovation Activity. The outcome of EIT Digital's API Assistant Innovation Activity is STAnD.

In the following, I explain the problem that STAnD is trying to address together with a high-level architecture view (Section 7.1.3.1) and Section 7.1.3.2 describes our contributions within the STAnD project, namely: (i) the integration of our wizard-based approach to help app developers with secure implementation of APIs for identity management; and (ii) a security service for analysis of identity management solution checking the proper usage of the identity plugin by app developers.

7.1.3.1 Activity Overview

API usage is continuously rising, and software API has become business enablers leading enterprises towards building more and more API with web/mobile native applications as the primary use case, empowering businesses to build more dynamic web/mobile native applications. APIs are not new, but they are relatively relevant in the Internet's age where the web/mobile native applications are growing and evolving. It has to be considered that by using APIs, web/mobile native application developers (and Companies behind them) may inadvertently open up the door to undesired access to all of their corporate data, making it a probe to potential vulnerabilities. A good example is illustrated by identity management APIs, which web/mobile native application developers use to provide a frictionless user experience for authentication and authorization. However, secure integration of these APIs within web/mobile native applications is challenging for web/mobile native application developers, who are not usually security experts. Figure 7.4 represents the overall high-level architecture regarding the STAnD project, which consist of three

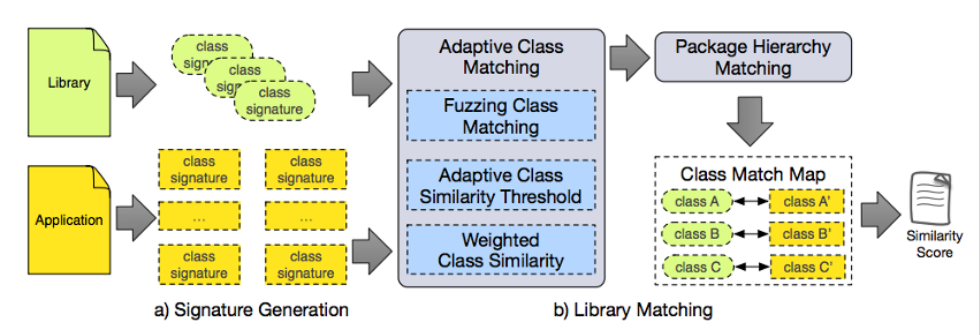


Figure 7.5: LibPecker Third-Party Library Detection Tool Architecture.

main components:

- User workstation with a browser represents the native app development environment and a browser. Through the browser, the user can access to STAnD web frontend.
- STAnD Site (Front- and Backend) provides an interface for native app developers to access the backend services and see the analysis results for their uploaded native apps.
- External Services enables native app developers to check their apps against known vulnerabilities and improve their code security based on the detailed reports related to the issues found in the code.

7.1.3.2 Contributions

In the following, I describe our main contributions in the STAnD project, which are integrating our wizard-based approach to help native app developers with secure implementation of multiple IdMPs scenario and security service for analysis of identity management solution checking the proper usage of the plugin by native app developers.

7.1.3.3 Identity Management Rest Service

The identity management rest service is intended to check whether the developer correctly used the Identity Management plugin or he/she has been made some changes to the integrated code, which may lead to breaking the security enforcement of the BCPs that is set out in the RFC-8252-OAuth Best Current Practices for the Native Apps [DB17]. To achieve this goal, we used an open-source tool called LibPecker [Zha18a] to detect third-party libraries in Android apps with high precision. App developers can customize the library code during the integration process within Android apps, and app developers use code obfuscators to eliminate dead library

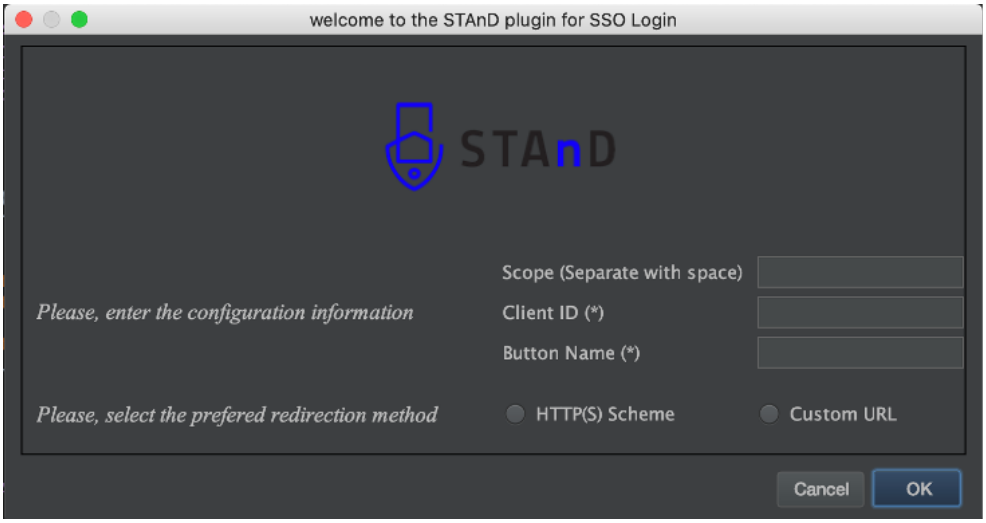


Figure 7.6: STAnD Identity Management Plugin GUI.

codes. LibPecker, in comparison with other state-of-the-art detection library tools, is an obfuscation resilient, highly precise, and reliable library detector for Android apps. LibPecker has been adopted signature matching methods, which enables it to generate a similarity score between a given Library and the app. The tool generates a separate signature for each class by fully utilizing the library's internal class dependencies. To equilibrate the effect of library code customization and elimination on the final similarity score calculation, LibPecker applies adaptive class similarity threshold and weighted class similarity score when calculating the final library similarity score. As shown in Figure 7.5, it consists of two major parts: 1) signature generation for all library classes and app classes, and 2) library matching process to give a similarity score between a given library and a given app based on the class signature.

The identity management rest service uses LibPecker as a backbone to check whether the identity management plugin was being used correctly or not. Therefore, it will take as an input app developer's app APK file and compare it with the AppAuth library to check whether app developers make some changes to the library code or not. As an output, the service provides the similarity level of the app with the library. If the app's similarity value with the library is more than 80%, we consider it a successful usage of the identity management plugin within app developer's app; otherwise, it is considered a failure.

7.1.3.4 Identity Management Plugin

To support app developers willing to integrate identity management APIs within their apps in an automated and secure manner, we integrate a simplified version of our wizard-based approach described in Chapter 6 within the STAnD tool. The identity management plugin provides a GUI

to support app developers in selecting the proper flow and configurations. Based on app developer's preferences, the plugin automatically integrates the customized and secure code within the app developers' app.

App developers need to install the plugin and select the scenario (Access delegation or Single Sign-On Login). Then, they interact with plugin GUI (Figure 7.6) to provide the necessary information and the secure code is automatically integrated within the app developer's app. In particular, the plugin does not have a list of IdMPs that app developers can select in the GUI, and some tasks include adding the dependencies, and the IdMP configuration should be done manually.

Chapter 8

mIDAssistant iGov Extension

Recently, because of the increasing access to Internet via mobile, many e-government projects are considering to transition from implementations that are not supporting the native scenario (SAML [LL09]) to the ones that are supporting the native scenario in governmental domains (iGov profile [ID18a, ID18b]). Although the iGov [ID18b, ID18a] documents aim to assist native app developers with secure integration of iGov solutions, if developers are not security experts and do not understand security critical code, their solutions may be exposed to serious security vulnerabilities.[1]

To this end, we reuse and further extend our mIDAssistant plugin to support the iGov profile based on the rational reconstruction of the iGov profile in Section 3.3.2.1. In this way, native app developers have to be worried about neither the core functionalities nor the enforcement of BCPs as both of them are handled by our mIDAssistant_iGov plugin.

Section 8.1 explains the main changes in the wizard DB population and wizard questions to satisfy the requirements of iGov scenario. The implementation details besides a security analysis on the integrated code by the plugin are provided in Section 8.2. Finally, Section 8.3 provides some insights concerning the mIDAssistant_iGov application to support a scenario illustrated by the Italian digital identity framework SPID [AGI19b].

8.1 Wizard-based Approach Modifications

In this section, we described what is the needed information to be stored in the DB for each iGov IdMP (*DB Population*) and explain how our wizard-based approach customizes the wizard questions based on the client registration methods supported by an iGov IdMP (*Wizard Questions*).

[1] This chapter is based on our published work in [SCSR20].

8.1.1 DB population

We explain which information should be provided by security experts for each new iGov *IdMP*.

1. *IdMP*.`Redirect` specifies the redirection mechanisms supported by the *IdMP* and can assume the following values: `HTTPS` if only HTTPS URI redirection is supported; `CustomURI` if only Custom URI Scheme is supported; and `both` if both of them are supported.
2. *IdMP*.`Acr_Values_necessity` specifies whether the level of assurance query parameter `acr_values` is mandatory for the *IdMP*, and can assume the values `Yes` | `No`.
3. *IdMP*.`Acr_Values` specifies the supported level of assurance (i.e. the values of the `acr_values` query parameter), and can assume `LoA1` | `LoA2` | `LoA3`.
4. *IdMP*.`Discovery` specifies the discovery URL, where the *IdMP* publishes its metadata (such as endpoints and scopes supported).
5. *IdMP*.`Scenario` specifies the supported scenarios, and can assume the following values: `SSO` if only Single Sign-On Login is supported; `AD` if only Access Delegation is supported; and `both` if both of them are supported.
6. *IdMP*.`Client_Registration_Methods` specifies the supported client registration method and assumes the values `Static CR` | `Dynamic CR` | `both` .
7. *IdMP*.`Profile` specifies the supported profile and assumes the values `Core` | `iGov`.

Among the information stored in the DB, `Acr_Values_necessity`, `Redirect`, and `Client_Registration` are used by the wizard to customize the questions for the app developer (as described in Section 8.1.2), while the discovery URL is used to properly configure the AppAuth SDK that is integrated within our plugin to enforce BCPs.

8.1.2 Wizard Questions

In Listing 8.1, we show which are the questions asked to a *C* app developer by the wizard (in *italic*) to support the iGov scenario. In addition, Listing 8.1 explains how the questions are customized according to the information provided in the *DB Population* process. According to the choices of the app developer, the wizard automatically integrates the code incorporating the security and privacy considerations (discussed in Section 3.3.2) within the app.

Listing 8.1: Wizard Questions for iGov Scenario.

```
// General IdMPs Selection
1. Please, select IdMP among the list of supported Providers
if the IdMP.Profile=iGov then
The wizard reads the endpoints for the selected iGov Provider from the IdMPs DB, fills the AppAuth conf file, and populates the questions within the plugin GUI.
// Implementation Selection
If the IdMP.Client_Registration_Method=both then
2. Please, select the implementation methodology: either [2.1] iGov following Static CR, or [2.2] iGov following Dynamic CR
Otherwise, ignore the question and go to Configuration.
// Configuration
3. Please, enter the configuration information: Scopes and the Name of the Button you create
If [2.1] or IdMP.Client_Registration_Method= Static CR then Enter ClientID
If IdMP.Acr_Values_necessity=Yes then Enter Acr_Values
The provided info are used by the wizard to update the AppAuth conf file and initialize the login button.
// Redirection
If IdMP.Redirect=both then
4. Choose the preferred Redirection Method: either [4.1] HTTPS scheme, or [4.2] CustomURI scheme
If [4.1] or IdMP.Redirect=HTTPS then Enter your valid domain URL: scheme, host, and path
If [4.2] or IdMP.Redirect=CustomURI then Enter the Custom URI of your app
The provided info are used by the wizard to complete the AppAuth conf file and fill the Intent filter in the Android manifest.
```

8.2 Implementation and Experimental Analysis

To assess the practical value of our approach, we extend the mIDAssistant plugin [SCRS19] to support app developers with secure and compliant code integration of the iGov flows. The mIDAssistant_iGov plugin has been developed using the Intellij Idea environment,[2] it can be integrated in Android Studio and leverages the AppAuth library [OID16] as the main building block to guarantee the enforcement of BCPs. While the iGov profile following static client registration is already implemented in the plugin, the iGov profile following dynamic client registration and the extensions required in SPID are in progress. In the following, we explain the workflow of the mIDAssistant_iGov plugin and an evaluation of the code integrated by the plugin in terms of security.

[2] https://www.jetbrains.com/idea/

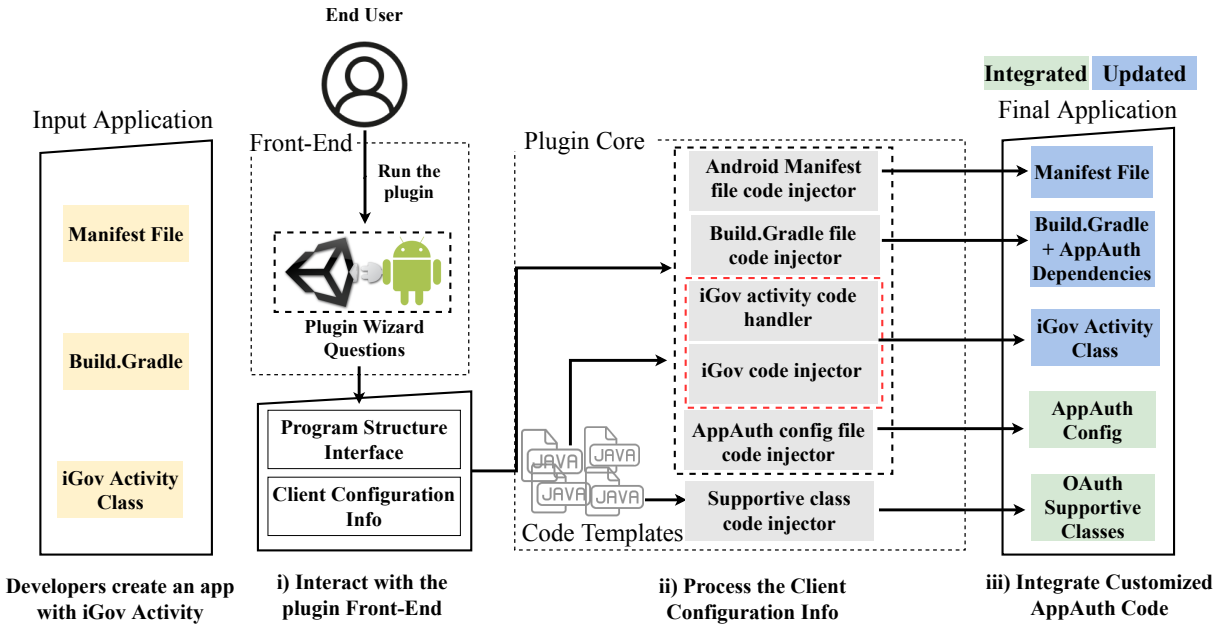


Figure 8.1: mIDAssistant iGov Plugin Workflow.

8.2.1 mIDAssistant_iGov Plugin

Assuming that an app developer creates a *C* app with an iGov activity beforehand, as shown in Figure 8.1, the mIDAssistant_iGov plugin workflow contains three main steps, namely: (i) interacting with the plugin front-end, (ii) processing the obtained information from the plugin front-end in the core, and (iii) integrating the customized AppAuth code within the app developer's app. In Step (i), developers run the mIDAssistant_iGov plugin and interact with the plugin front-end to compile wizard questions as explained in Listing 8.1. This step provides client configuration info and the Programming Structure Interface (PSI) as an output. The aforementioned outputs are needed for the code integration process within the plugin core. After that, in Step (ii), the plugin utilizes the code templates together with the inputs from the previous step (client configuration info and PSI) to initialize the code injector/handler within the plugin core. While PSI is used in the plugin core by the code injector/handler to find the specific file element for the code integration process, the client configuration info is used to instantiate the customized code. It is worth mentioning that the code templates are mostly obtained from the AppAuth [3] repository. In the following, we explain the functionality of each code injector/handler component within the plugin core. In case developers choose the HTTPS redirection scheme within the wizard GUI, the *Android Manifest file code injector* extracts the HTTPS redirection scheme value from the client configuration info to initialize the customized code to handle the HTTPS redirection. The *Build.Gradle file code injector* initializes the customized code to add the AppAuth

[3] `https://github.com/openid/AppAuth-Android`

SDK dependencies. In addition, in case of Custom URI selection by developers, it handles the Custom URI redirection by obtaining its relevant value from the client configuration info. The *AppAuth config file code injector* initializes a JSON structure by extracting the values from the client configuration info. This step is necessary to enable the app communication with specific iGov IdMPs. The *iGov code injector* and *iGov activity code handler* initialize the customized code to add the main iGov functionality within the iGov activity class. They accomplish the aforementioned task by obtaining the relevant values and code from the client configuration info and code templates, respectively. While *iGov activity code handler* adds the initialization code for the iGov button within the `OnCreate` method in the iGov activity class, the *iGov code injector* provides the main functionalities in the above-mentioned class for the iGov button. In contrast to previous code injectors, the *Supportive class code injector* initializes the customized code only from code templates (Figure 8.1) which provides necessary iGov functionality, such as read/validate the app configuration JSON file. Finally, in Step (iii), the code injector/handler incorporates the customized code into the app to update Build.Gradle, Android Manifest, and iGov activity class files. It also integrates the newly created JSON config and supportive classes files.

The plugin core integrates necessary files to enable app communication with specific iGov IdMPs and leverages the AppAuth SDK as a main core to ensure the enforcement of BCPs for native apps. Therefore, our solution helps developers eliminate the maze of the OIDC/iGov related documents and the corresponding security considerations to avoid wrong implementation choices. In addition, as the plugin takes care of the iGov integration, developers can focus on other tasks which are out of the scope of the plugin. For example, developers should write the code to parse the JSON response from the *UserInfo* endpoint; we decide to delegate this task to developers as they should decide what they need to parse in the JSON response. In addition, in the case that iGov profile with DCR is selected, the developer has to perform the registration procedure that is composed of the following two main tasks: (i) complete the code to call the DCR endpoint by adding the `access_token` obtained from iGov IdMPs during the registration phase; and (ii) write the code to register the public key with the iGov provider and to store the private key securely. We delegate the aforementioned tasks to developers to avoid the sharing of `access_token` and public/private keys of developers with the plugin. Indeed, if we obtain these values within the plugin, developers have to fully trust us that we do not misuse the obtained values. It is worth mentioning that the plugin helps the developer also with these tasks by adding comments into the places where the developer must complete the code.

Besides the advantages provided by our solution, there are two main limitations: there is a need for (i) integrating new BCPs within the plugin whenever new threat models and security considerations have been published; and (ii) keeping the IdMP DB up-to-date. The possible solutions for problem (i) can either involve experts from the OAuth working group or have dedicated security experts (in the OAuth/OIDC field), who are capable to analyze the new RFCs and integrate it with the plugin. Problem (ii) is easier to handle as we need to periodically inspect the provider's website to detect potential changes and integrate them within the plugin DB through a plugin update release.

To increase the trust in the correctness of the plugin, we are going to make its code open-source.[4] This provides a way to perform a code-review process by the experts in this field. In addition, as detailed in the paragraph below, we have used both open-source and commercial static code analysis tools to perform the security source-code analysis on the code integrated by the plugin.

8.2.2 Security Analysis

In this section, we summarize the results of the security analysis performed on the code integrated by the mIDAssistant_iGov plugin. In our security analysis procedure, we consider the AppAuth SDK source code secure based on the following assumptions: AppAuth is an open source project primarily developed by the Google Identity Platform team and endorsed by the OIDC foundation since 2016; and it is used within Google's own client-side authentication stack in Android and reviewed by the Google's internal security team for vulnerabilities.[5] Given that, we verify only the security of the integrated code within the app developer's app by using both open-source and commercial source code analysis tools.

In the first phase, we perform a security analysis on the integrated plugin code by utilizing two open-source tools, namely: Super and MobSF [Sup18, Ope18]. The security analysis results reveal two possible security issues: (i) weak cryptography algorithm; and (ii) security information disclosure. With further analysis, we discover that both results are not security issues in our case. Indeed, the detected weak algorithm is used to create a hash of the configuration file that the SDK uses to detect the change in the configuration file to clear the authorization state. The second item is not a security issue, as the MobSF tool marks all the plain strings with a certain length as possible secret keys. In the second phase, we perform the source code analysis by utilizing a commercial tool called Approver [VAMC18]. The security report of the tool contains the following issues: (i) insufficient code obfuscation; and (ii) undesired access to private data. Although all the reported findings are security issues in general, they are considered out of the scope of our plugin work. Indeed, the main goal of our plugin is to help app developers with the integration of iGov flows within their app, therefore, we delegate the tasks for performing code obfuscation and encrypting the information in preferences to developers.

8.3 iGov Application on SPID

This section presents an application of iGov profile in the governmental and public administration use case, which is represented by the Italian digital identity framework SPID [AGI19b].

[4] `https://github.com/stfbk/mIDAssistant_iGov`

[5] The internal review was not published by Google, this information was obtained from a conversation with the main developer of AppAuth SDK.

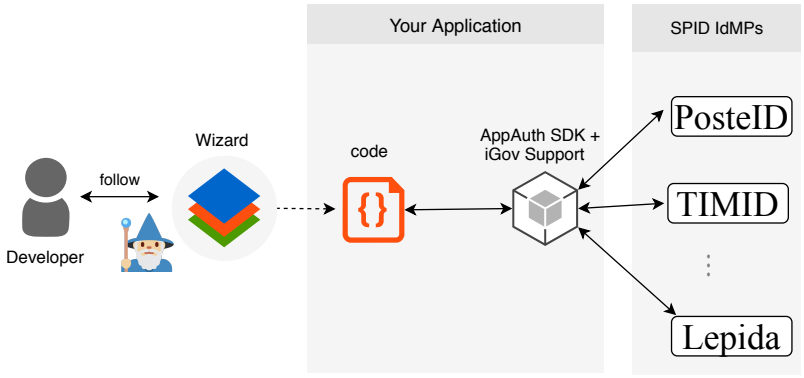


Figure 8.2: Proposed Approach to Support iGov (SPID Scenario).

While the SPID implementation is currently based on SAML, they want to migrate into an OIDC implementation of iGov profile (Section 2.4.1). Note that SPID has a customized implementation of the iGov profile, as it is mentioned in the SPID documentation [AGI19a], that combines both static and dynamic iGov native flows (See Section 3.3.2.1).

Considering Figure 8.2, app developers may face the same challenges as reported in Section 7.1.1 (multiple IdMPs integration) to implement the SPID scenario within their apps. Thus, to overcome the implementation challenges, app developers can use the mIDAssistant_iGov plugin to automates most of the SPID integration process and also enforces the BCPs concerning the iGov flow for native apps.

In this way, our approach allows app developers to integrate SPID iGov solutions effortlessly and in a secure manner. In particular, app developers simply interact with the mIDAssistant_iGov GUI to select their preferred IdMP and to provide the mandatory information. After that, the plugin automatically integrates the secure customized code within the app developer's app.

To integrate the support for SPID in mIDAssistant_iGov, it demands the following changes in the DB population process and wizard questions (Section 8.1).

In the DB population process :(i) Since in SPID the usage of `acr_values` is mandatory, *IdMP*.`Acr_Values_necessity` is always equal to `Yes`; (ii) as SPID introduces the `prompt` query parameter, the database should store its possible values. *IdMP*.`Prompt` can assume the values `Login` | `Consent` | `Login Consent`; and (iii) *IdMP*.`Profile` assumes the new value `SPID`. Finally, while in the general iGov IdMP scenario security experts are in charge of the IdMP DB population, the solution we envisage for SPID is to delegate this task to the Italian institution responsible for coordinating the implementation process of SPID, namely Agenzia per l'Italia digitale (AGID).

Concerning the wizard questions, the code integration scheme for the wizard is similar to Listing 8.1, apart from the following main differences: (i) the "*Implementation Selection*" question (cf. Line 7) is omitted, as SPID combines both Static and Dynamic client registration; (ii) the

"*Configuration*" questions (cf. Line 10) are slightly different, as the SPID profile contains two additional query parameters (`prompt` and `claim`) and `acr_values` appears as mandatory; (*iii*) in SPID, the developer must provide the configuration info for all the SPID providers (9 in the current federation), in a way that the user can choose the one he/she has already registered with.

Note that as mentioned in Section 8.2, the iGov profile following dynamic client registration and the extension required in SPID are postponed to future work. The main reason is that the following documents [ID18b, AGI19a] are not the final versions and there is still open discussion in OAuth WG to finalize the draft.

Chapter 9

Related Work

In this chapter, we present the related work. In particular, we compare our work to existing publications, considering the following criteria: IdM plugin competitors, approaches for detecting implementation flaws, threat modeling research papers, wrong AD/SSO implementation detection tools, and privacy-preserving solutions.

9.1 IdM Plugin Competitors

For IdM, there exist platforms (e.g., Auth0[1]) that play the role of an identity hub, sitting between the native app and multiple IdMPs. This kind of solution, similarly to our approach, aims to simplify the effort of native app developers. Yet, native app developers have to fully trust them (because, in principle, they can impersonate the user on the other IdMPs). On the contrary, in our solution, the native app directly interacts with each IdMP, without leveraging an intermediate third-party service. Concerning other tools supporting AppAuth, it is worth mentioning that Xamarin[2]—a tool for cross-platform mobile app development—provides an SDK (Xamarin.OpenID.AppAuth) for helping developers to communicate with OAuth and OIDC Providers. As a consequence, by leveraging the "Xamarin.OpenID.AppAuth" SDK, our approach can be applied on top of Xamarin so to support the cross-platform scenario as well.

[1] `https://auth0.com`

[2] `https://dotnet.microsoft.com/apps/xamarin`

9.2 Approaches for Detecting Implementation Mistakes

Many tools have been proposed to help native app developers to implement secure code, both just after the development cycle, such as Chex [LLW+12] and RoleCast [SMS11], and during the implementation phase, such as Android Lint Tool [Goo16], FixDroid [NWA+17], and App Link Assistant [Goo17]. Among them, App Link Assistant provides a wizard-based approach to help developers with proper configuration of HTTPS scheme within their native apps that is a crucial step within an SSO or AD implementation to secure the HTTPS redirection. Compared to this tool, our method assists developers in the whole implementation process, including HTTPS proper configuration. In addition, as the existing supportive tools are aimed to address specific native app quality or security problems, they are not suitable for dealing with issues specific to SSO or AD implementations.

9.3 Threat Modeling Research Papers

In addition to the developer-assistant tools, other studies exist to help developers understand possible vulnerabilities in the implementation of OAuth/OIDC solutions. In the following, we first summarize the OAuth/OIDC threat modeling papers that help native app developers with getting familiar to the possible impact of wrong implementations, and later we detail a list of research tools capable of detecting vulnerable OAuth/OIDC implementations within native apps.

Liu et al. [LLWZ18] performed a comprehensive study of OAuth *Implicit* and *Authorization Code* flows by detailing their differences on the web and mobile scenario and summarizing possible security issues on the implementation of the OAuth code flow within native apps. The improper storage of `client_secret` and `access_token` in plain text, using an embedded browser as *UA*, incorrect usage of authentication proof, handling redirection in native app improperly and lacking transmission protection have been reported in their work as common security faults in the OAuth implementation for native apps. A complete threat modeling of OIDC specification and its current implementations have been presented by Navas and Beltrán in [NB19]. They have identified up to 16 different security and privacy attack patterns (e.g., token reply, code reply and fake token generation) that are described in detail within their work. Furthermore, possible mitigations and solutions are proposed for both implementation and specification aspects.

9.4 Wrong AD/SSO Implementations Detection Tools

Wang et al. [WZL+15] proposed a semi-automated framework to assess AD implementation in Android native apps called AuthDroid. AuthDroid combines automatic static source code with manual dynamic traffic analysis to find vulnerabilities in AD implementations, such as improper

UA usage, inadequate transmission protection, insecure secret management, problematic server-side validation and wrong authentication proof. In their studies, they considered 15 main Chinese IdMPs and 4151 apps that are downloaded from the Chinese app market. The results proved that from 1372 apps that are integrating at least one of the OAuth IdMPs, 86% percent are vulnerable. In addition, 13 out of 15 IdMPs are enforcing embedded browser as *UA* and only 4 out of 15 enforces HTTPS to protect the communication. While AuthDroid can detect most of the potential vulnerabilities in faulty AD implementations among our interest, we cannot use AuthDroid in our experimental analysis phase as the tool is neither open-source nor available online. It is worth mentioning that we contacted AuthDroid authors for their tool availability and by the time of writing this thesis, we have not received any notification yet. In 2017, Zuo et al. [ZZL17] proposed AuthScope, an automated tool to find vulnerable authorization mechanisms on the server-side for apps integrating login with Facebook. AuthScope is automatically running the app under test to trigger the login and then analyze the intercepted app traffic to find the key query parameters in the request/response for performing vulnerability analysis. They analyzed 4836 apps from top 10% of Google Play Store, among them 2976 have been detected by the tool as vulnerable. Further analysis performed by the authors proved that only 597 cases are actually vulnerable and the other cases are false positives.

An OAuth Fuzzer has been developed by Yang et al. in [YLS17] to test SSO implementations of 600 top-ranked Google Play Store and Chinese App Store apps that are integrating Google, Facebook, and Sina as IdMP. They performed a deep static code analysis on the SDKs that are provided by the IdMPs. Based on the obtained knowledge, they performed dynamic traffic analysis by leveraging the developed fuzzer to check the implementation against wrong identity proof, lack of verification on the identity proof, and other known vulnerabilities, such as improper *UA* usage and app secret disclosure. The result shows that 41% of apps under test are using the wrong identity proof to authenticate users. Recently, an automated blackbox tester for SSO vulnerabilities called MoSSOT has been developed by Shi et al. [SWL19]. MoSSOT is a model-based blackbox testing that is built on top of pyModel to take Finite State Machines (FSM) as an input and generate test cases to find possible SSO vulnerabilities. Indeed, they intercepted the app traffic to find the key query parameters and then substitute the query parameters based on the test cases to find possible vulnerabilities. They analyzed 550 apps from two third-party app stores: Apkpure and Wandouija. The analysis result shows that 72% of the apps under analysis are vulnerable to at least one of the following security issues, which are: `access_token` replacement, insecure `client_secret` and `access_token` management, code reply and profile vulnerability. Rahat et al. [ARFT19] proposed a tool—OAuthLint—to check secure implementation of SSO solutions within native apps in the Google Play Store. OAuthLint leverages a query driven static analysis method to automatically check the implementation of native apps against known vulnerabilities, such as locally bundled `client_secret`, improper *UA* usage and storing `access_token` in plain text. They analyzed 600 apps from the Google Play Store, 32% of them are vulnerable. Although OAuthLint is capable of detecting attacks that are among our interests, the tool is neither open-source nor available online. In addition, we contact OAuthLint authors

for the availability of their tools and they informed us the tool will be released online soon.

While the previous research papers focus on the native scenario, I summarize some of the available works on AD/SSO detection tools for the web scenario. A free Chrome browser extension called "OAuthGuard" is introduced in [LMC19]. OAuthGuard is an AD/SSO vulnerability scanner capable of detecting CSRF attacks, user impersonation, authorization flow misuse, unsafe `access_token` transfer, and privacy leaks within web applications that are using Google as IdMP. It is worth mentioning that their tool cannot detect complex attacks that demand in-depth server-side analysis (e.g., Mix-Up attack). In their studies, they considered top-ranked 1000 websites and the results proved that from 137 websites using the google SSO/AD solution, 69 are vulnerable to at least one attack.

Mainka et al. [MMSW17] propose a tool—PrOfESSOS— to check the secure SSO implementation within certified OIDC libraries. Their approach simulates both honest and malicious Identity Provider in order to be able to perform more sophisticated attacks. Therefore, their tool can control more SSO-related messages, remove or manipulate query parameters, and perform more complex attacks. The authors categorize the attacks into two main categories: (i) single phase; and (ii) cross-phase attacks. While the former represents attacks that demand a query parameter manipulation (e.g., replay attack), the latter demands a complex attack configuration and manipulates multiple messages distributed along with the whole protocol flow (e.g., Mix-Up attack). In their studies, they perform analysis over eight certified OIDC libraries. The results proved that 75% (6 out of 8) are vulnerable against single-phase attacks, and all of them are vulnerable against cross-phase attacks.

SSOScan has been developed by Zhou et al. in [ZE14] to check implementation flaws in the web application, which are integrating Facebook SSO/AD APIs. SSOScan takes the website URL as an input and firstly checks for the FaceBook login button on the website. If the previous step is successful, the tool simulates some attacks on the website while obtaining responses and monitoring the network traffic to determine the attack feasibility. Their tool is capable of detecting `access_token` misuse, OAuth credential leakage, and signed_request misuse. Their studies have analyzed more than 1600 websites, and 20% of them suffer from at least one vulnerability that the tool can detect. Their method suffers from the following limitations: (i) SSOScan can only detect recognizable attacks based on either traffic monitoring or simulating predictable user events, and (ii) SSOScan cannot detect vulnerabilities that need heavy server-side scanning and complex user interaction.

Calzavara et al. [CFM$^+$18] propose a browser extension—WPSE—for security monitoring of web protocols to ensure their compliance with the intended protocol flow. WPSE can automatically protect the users of vulnerable websites against a large class of attacks. In order to enable WPSE to check the website implementation against possible vulnerabilities, the authors define the web protocols in terms of the HTTP(S) exchanges observed by the web browser. Note that, the specifications can be written once and enforced on several sites. In their studies, they check SSO implementations of 90 website and find 55 website with security flaws. It is worth men-

tioning that by using their approach, they found a novel attack on the google implementation of SAML 2.0.

Finally, in 2021, Philippaerts [Phi21] proposes OAuch, a security best current practices and threats analyzer for AD/SSO server implementations. The tool takes the IdMP metadata as an input and performs a large set of security-related test cases to uncover relevant threats with regard to their implementation beside pointing out possible security improvements. The test cases are designed based on the original OAuth documentation [Har12] and several other documents that summarize the best current practices for OAuth/OIDC implementations [SBJ+14, LBLF21, LMH13].

9.5 Privacy-Preserving Solutions

In [WW17] authors propose an approach to enforce privacy goals: *confidentiality*, *interveinability*, *transparency* and *accuracy*. Users encrypt their personal data before sending to IdMP, so it gives the possibility of selective disclosure as only user can open them for applications. It provides a validation Process, where IdMPs validate user's attributes. The validation can be done by any Civil registration agencies, banks, and other governmental and non-governmental agencies which issue user data.

EL PASSO [ZKS+20] offers an easy-to-deploy (for both IdMP and controller) asynchronous and unlinkable authentication. It fulfills the following privacy goals: *data minimization, interveinability, user accountability* and *deployment* by avoiding synchronous communication between controllers and IdMPs. It prevents both controller and IdMP to track users, allows selective disclosure of personal data, holds misbehaving users accountable in cooperation with law enforcement authorities.

Hammann et al. [HSB20] proposed an extension for the OIDC to prevent Identity Providers from user tracking within different clients and also protects the user from tracking by colluded clients by masking the client identifier (`sub`) in the user agent, and by providing a new design for the pairwise element, respectively. In order to mask the client identifier, they utilize a simple cryptographic hash function and within the design of pairwise element, they utilize the zero-knowledge proof. It is worth mentioning that the proposed method provides the same level of protection against colluded clients as the standard pairwise in the presence of additional user-identifying information and it only protects the user tracking from Identity Provider. However, the proposed method can be easily adopted on top of the OIDC standard to protect the user against Identity Provider. Given that, they started a collaboration with the OpenID Foundation to provide a reference implementation.

Wanpeng et al. [LM20] performed a systematic analysis of the user access privacy in the OAuth and consequently, OIDC standards. The analysis revealed that to make the protocol thoroughly

privacy-friendly, it demands fundamental changes to how they operate. The scope of their analysis is very similar to the work proposed by Hammann [HSB20] as they investigated the ways Identity Providers can track users between varied clients during login and user information retrieval. The authors claim that the problem lies in the OAuth/OIDC specifications as they do not consider privacy in their design. They highlight that the main cause of user tracking by Identity Providers within varied clients is due to the pre-registration phase of clients at Identity Provider that reveals security-critical information about clients, such as: `client_id` and `redirect_uris`. These two query parameters are enabling the Identity Provider to identify clients that users are trying to connect. Furthermore, they criticized the audience claim in *access_token* and *id_token* as they reveal the identity of the client. However, the authors agreed that the presence of audience claim is necessary to avoid crafted tokens. Thus, they suggest a client-based mitigation strategy similar to the one presented in [HSB20], which acts as an intermediary between clients and Identity Providers to transmit the authorization requests/responses.

Authors in [PM12] surveyed the current approaches for addressing privacy in identity management systems. They list privacy requirements and discuss how the currently deployed systems address them. They state an identity management system need to address the following to be privacy-friendly: (i) trust model; (ii) multiple unlikable identities; (iii) selective disclosure; (iv) consent; and (v) privacy-respecting sharing of personal information. Their analysis on SAML 2.0 reveals that the system does not support selective disclosure by default as it depends on IdP for providing a user interface to define selective disclosure policy. On the other hand, CardSpace system provides a mechanism to reduce the trust requirement on the IdPs and consequently avoid monitoring user's activity by IdPs. eID is the most recent identity management system that is the electronic equivalent of citizens' national identity cards. However, except the German eID system, all the others deployed systems failed to provide a complete privacy-preserving solution due to revealing more information than is strictly necessary. They claim that OIDC does not support any degree of anonymity or pseudonymity due to the issuance of a global identifier for user authentication; hence, it is much less privacy-friendly.

Morkonda et al. [MvOC21] investigated the users privacy implications of OAuth implementations within top-ranked websites that integrate Google, Facebook, Linkedin, and Apple as IdMPs. To do so, they developed OAuthScope that is capable of extracting OAuth request query parameters. It identifies areas to improve user's privacy (e.g., consent screen interface) However, some of their recommendations are already adopted by IdMPs, e.g., requiring to justify requested scopes.

What these works have in common is that they either: (i) demands fundamental changes to the way that protocol operates; and/or (ii) proposes browser-based solutions or adding cryptographic functions. They require major changes in IdMPs and applications' current implementation, time to adapt, and computational overhead. Whereas our methodology utilizes existing OAuth/OIDC features and It cannot prevent honest but curious IdMP attack, in any case.

Chapter 10

Conclusion and Future Work

In this thesis, to verify the applicability and the role of OAuth/OIDC BCPs to increase the security of IdMP solutions and consequently their effect on the secure integration of IdMPs' solution within native apps, we performed a survey on popular IdMPs and top-ranked Google Play Store apps to check their compliance against the aforementioned BCPs. To this aim, we first studied the relevant OAuth/OIDC documentations related to native apps to extract the relevant BCPs for native apps and then defined a selection procedure for popular IdMPs and top-ranked Google Play Store apps. Our BCPs compliance analysis of popular IdMPs represented that 50% of IdMPs under analysis did not follow BCPs, and their online documentations were misleading for native app developers. In addition, we analyzed 80+ Google Play Store top-ranked apps that integrate at least one of the IdMP's understudy to check their compliance with BCPs. Our analysis revealed that 86% of app instances with official SDK and 100% of app instances with unofficial SDK are not fully compliant with BCPs. It is worth mentioning that 36% of app instances contain native app developer or IdMP faults. This confirms the difficulty in digesting OAuth/OIDC documentations and its security related documents. Based on our analysis, the current situation concerning the Google Play Store apps is the result of the following issues: (i) sparseness of OAuth/OIDC BCPs, as they have been published gradually over the years, (ii) wrong interpretation of OAuth/OIDC documents, as native app developers are not (necessarily) security experts, (iii) lack of BCPs adoption by IdMPs that lead to difficulty in the integration of AppAuth SDK within native apps and even in case of compliant IdMPs, native app developers need to write a secure code to invoke the SDK within their native apps, which is a daunting task, and (iv) lack of performing a risk assessment procedure, as native app developers are usually under pressure to add new functionalities.

Given that, we proposed a methodology to assist native app developers with secure integration of OAuth/OIDC solutions through (i) providing a complete reference model for OAuth/OIDC solutions that paves the way for the tool-based approach and assists native app developers in the procedure of conducting risk assessment, and (ii) designing a wizard-based approach and imple-

menting it in a working prototype for the Android Studio environment (mIDAssistant). While the former assists native app developers to perform a risk assessment without being worried about its complexity as the reference model captures all the necessary information, the latter provides an automatic way to integrate the core functionalities related to the OAuth and/or OIDC solutions besides the enforcement of BCPs by leveraging the AppAuth SDK.

Finally, the effectiveness of our approach has been verified in a number of relevant real-world scenarios, research and innovation projects (e.g., the EIT Digital activity API Assistant) and in the context of industrial collaborations (Poste Italiane, IPZS). Furthermore, we had the opportunity to present our work to the OAuth working group experts (during the OAuth Security Workshop), and they have shown interest in our approach.

10.1 Future Work

The presented OAuth/OIDC risk assessment methodology, wizard-based approach and the mIDAssistant plugin can be straightforwardly generalized to support other use-cases, which we are currently doing. For example, in the case of mIDAssistant plugin, we are working toward an extension that assists native app developers with the integration of Bank account aggregators, which is currently becoming popular thanks to the opportunities offered by the revised Payment Service Directive (PSD2), which is a European regulation to regulate payment services and payment service providers throughout the European union and European economic area. To assess the usability aspect and how native app developers might use mIDAssistant plugin in the real world use cases, we plan to design and conduct a user-study with both students and native app developers.

In addition, to provide a tool-chain that can assist native app developers in the whole development life cycle, we plan to extend our work in two possible directions.

A first improvement is to provide a security-analysis-as-a-service that performs dynamic analysis on native apps to spot possible implementation flaws concerning their OAuth/OIDC solutions. In this way, native app developers can submit their native app into our service. The security analysis service will check their native app against known OAuth/OIDC-related implementation flaws and provide a report with the found issues and possible ways to mitigate them. Note that our BCPs compliance analysis (see Chapter 4) can be used as a starting point to define the security checks that our dynamic analysis service must perform. In this way, we aim to automatically perform the checks that were not fully automated in the current version of our BCPs compliance analysis.

The second improvement of our methodology will be to integrate the OAuth/OIDC risk assessment methodology and the wizard-based approach. In particular, the idea is to provide as an input of the mIDAssistant the selected choices of native app developers at the end of the risk assessment process. mIDAssistant will be thus configured according to the choices. In this way, native

app developers can perform a risk assessment procedure that enables them to make an informed decision w.r.t. their implementation choices by considering their constraint on both budget and time. In addition, we can give more flexibility to native app developers by giving them all the possible choices w.r.t. supported features by the IdMP and then mIDAssistant can automatically integrate the code that enforces the native app developers choices within their native apps.

Bibliography

[AC13] Andrea Avancini and Mariano Ceccato. Security testing of the communication among android applications. In *2013 8th International Workshop on Automation of Software Test (AST)*, pages 57–63. IEEE, 2013.

[AGI19a] AGID. Linee Guida OpenID Connect in SPID. `https://docs.italia.it/AgID/documenti-in-consultazione/lg-openidconnect-spid-docs/it/bozza/`, 2019.

[AGI19b] AGID. Sistema Pubblico di Identità Digitale. `https://www.spid.gov.it/`, 2019.

[Ama21a] Amazon. Login with Amazon Documentation. `https://developer.amazon.com/docs/login-with-amazon/android-docs.html`, 2021 (last accessed February, 2021).

[Ama21b] AmazonDev. Amazon Android SDK. `https://developer.amazon.com/it/sdk-download`, 2021 (last accessed February, 2021).

[ap18] ac pm. Inspeckage. `https://github.com/ac-pm/Inspeckage`, 2018.

[App18a] AppCritique. AppCritique. `https://appcritique.boozallen.com/`, 2018.

[App18b] AppKnox. AppKnox. `https://appknox.com/`, 2018.

[App18c] AppWatch. AppWatch. `http://appwatch.io/`, 2018.

[AR18] App-Ray. App-Ray. `https://app-ray.co/`, 2018.

[ARFT19] Tamjid Al Rahat, Yu Feng, and Yuan Tian. OAuthLint: An Empirical Study on OAuth Bugs in Android Applications. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 293–304, 2019.

[Aut21a] Auth0. Auth0 API Guide. `https://auth0.com/docs/api/info`, 2021 (last accessed February, 2021).

[Aut21b] Auth0Dev. Auth0 SDK for Android. https://auth0.com/docs/libraries/auth0-android, 2021 (last accessed February, 2021).

[BCPR20] Andrea Bisegna, Roberto Carbone, Giulio Pellizzari, and Silvio Ranise. Micro-id-gym: A flexible tool for pentesting identity management protocols in the wild and in the laboratory. In *International Workshop on Emerging Technologies for Authorization and Authentication*, pages 71–89. Springer, 2020.

[Bee21] Beeminder. Beeminder API Reference. http://api.beeminder.com/#beeminder-api-reference, 2021 (last accessed February, 2021).

[Bha18] Ashish Bhatia. A Collection of Android Security Related Resources. https://github.com/ashishb/android-security-awesome, 2018.

[BJM08] Adam Barth, Collin Jackson, and John C Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 75–88, 2008.

[Box21a] Box. Box API Guide. https://developer.box.com/docs/quickstart-guides, 2021 (last accessed February, 2021).

[Box21b] BoxDev. Box SDK for Android. https://github.com/box/box-android-sdk, 2021 (last accessed February, 2021).

[CBSL19] Brian Campbell, J Bradley, N Sakimura, and T Lodderstedt. OAuth 2.0 Mutual TLS Client Authentication and Certificate Bound Access Tokens (draft-ietf-oauth-mtls-17). *Internet Engineering Task Force (IETF)*, 2019.

[CFM+18] Stefano Calzavara, Riccardo Focardi, Matteo Maffei, Clara Schneidewind, Marco Squarcina, and Mauro Tempesta. WPSE: fortifying web protocols via browser-side security monitoring. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1493–1510, 2018.

[Cla17] Georgiu Claudiu. A Command Line Tool to Download Android Applications Directly from the Google Play Store. https://github.com/ClaudiuGeorgiu/PlaystoreDownloader, 2017.

[CMJG15] Brian Campbell, Chuck Mortimore, Michael B Jones, and Yaron Y Goland. Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants (RFC7521). *Internet Engineering Task Force (IETF)*, 2015.

[CPC+14] Eric Y. Chen, Yutong Pei, Shuo Chen, Yuan Tian, Robert Kotcher, and Patrick Tague. OAuth Demystified for Mobile Application Developers. In *Proceedings of the ACM SIGSAG conference on computer and communications security, CCS 2014, Scottsdale, AZ, USA, November 3-7, 2014*, pages 892–903, 2014.

[CPT+16] Eric Chen, Yutong Pei, Yuan Tian, Shuo Chen, Robert Kotcher, and Patrick Tague. 1000 Ways to Die in Mobile OAuth. https://youtu.be/h7m0JScYwss, 2016. Black Hat Information Security Event.

[Cuc18] CuckooDroid. CuckooDroid. https://github.com/idanr1986/cuckoo-droid, 2018.

[DB17] William Denniss and John Bradley. OAuth 2.0 for Native Apps. RFC8252. *Internet Engineering Task Force (IETF)*, 2017.

[DDFH+15] George Danezis, Josep Domingo-Ferrer, Marit Hansen, Jaap-Henk Hoepman, Daniel Le Metayer, Rodica Tirtea, and Stefan Schiffner. Privacy and data protection by design-from policy to engineering. *arXiv preprint arXiv:1501.03726*, 2015.

[Dro21a] DropBox. DropBox OAuth 2.0 Guide. https://www.dropbox.com/developers/reference/oauth-guide#oauth-2-flow, 2021 (last accessed February, 2021).

[Dro21b] DropBoxDev. Unified Cloud Storage API Android. https://www.dropbox.com/developers/documentation/java, 2021 (last accessed February, 2021).

[DSCR21] Salimeh Dashti, Amir Sharif, Roberto Carbone, and Silvio Ranise. Automated risk assessment and what-if analysis of openid connect and oauth 2.0 deployments. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 325–337. Springer, 2021.

[Fac21a] Facebook. Facebook Login for Android. https://developers.facebook.com/docs/facebook-login/android, 2021 (last accessed February, 2021).

[Fac21b] FacebookDev. Facebook Login for Android SDK. https://developers.facebook.com/docs/android/componentsdks, 2021 (last accessed February, 2021).

[FKS16] Daniel Fett, Ralf Küsters, and Guido Schmitz. A comprehensive formal security analysis of oauth 2.0. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1204–1215, 2016.

[FKS17] Daniel Fett, Ralf Küsters, and Guido Schmitz. The web sso standard openid connect: In-depth formal security analysis and security guidelines. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, pages 189–202. IEEE, 2017.

[FS18] F-Secure. Drozer. https://labs.f-secure.com/tools/drozer/, 2018.

[GDP16] REGULATION (EU) 2016/679 OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL of 27 April 2016, 2016.

[GDP19] Romanian National Supervisory Authority for Personal Data Processing (ANSPDCP). A new fine for the application of gdpr, 2019.

[Goo16] Google. Android Lint. https://developer.android.com/studio/write/lint, 2016.

[Goo17] Google. App Link Assistant Tool. https://developer.android.com/studio/write/app-link-indexing, 2017.

[Goo21] Google. OAuth 2.0 for Mobile and Desktop Apps. https://developers.google.com/identity/protocols/OAuth2InstalledApp, 2021 (last accessed February, 2021).

[Har12] Dick Hardt. The OAuth 2.0 Authorization Framework (RFC6749). *Internet Engineering Task Force (IETF)*, 2012.

[HiD21a] HiDrive. HiDrive OAuth 2.0 Authentication Guide. https://developer.hidrive.com/basics-flows/, 2021 (last accessed February, 2021.

[HiD21b] HiDriveDev. HiDrive Android SDK. https://github.com/HiDrive/hidrive-android-sdk, 2021 (last accessed February, 2021).

[HJDA18] Phil Hunt, Mike Jones, William Denniss, and Morteza Ansari. Security Event Token (SET) (RFC8417). *Internet Engineering Task Force (IETF)*, 2018.

[HJR15] Marit Hansen, Meiko Jensen, and Martin Rost. Protection goals for privacy engineering. In *IEEE SPW*, 2015.

[Hof19] Xenia Hofmeier. Formal analysis of web single-sign on protocols using tamarin. 2019.

[HSB20] Sven Hammann, Ralf Sasse, and David Basin. Privacy-Preserving OpenID Connect. In *Proceedings of the 15th ACM ASIACCS*, 2020.

[IBM18] IBM. IBM Application Security. https://appscan.ibmcloud.com/AsoCUI/serviceui/home, 2018.

[IBM21a] IBMCloud. IBM Cloud Identity Connect SSO Configuration for the OpenID Connect Provider. https://www.ibm.com/support/knowledgecenter/SSCT62/com.ibm.iamservice.doc/tasks/oidc_app_sso.html, 2021 (last accessed February, 2021).

[IBM21b] IBMCloudDev. IBM Verify Android SDK. https://github.com/IBM-Security/verify-sdk-android, 2021 (last accessed February, 2021).

[ID18a] Internet-Draft. International Government Assurance Profile (iGov) for OAuth 2.0, 2018.

[ID18b] Internet-Draft. International Government Assurance Profile (iGov) for OpenID Connect 1.0, 2018.

[Imm18] Immuniweb. Immuniweb. https://www.htbridge.com/immuniweb/, 2018.

[JAD18] JADX. JADX. https://github.com/skylot/jadx, 2018.

[JBM15] Mike Jones, John Bradley, and Maciej Machulak. OAuth 2.0 Dynamic Client Registration Management Protocol (RFC7592). *Internet Engineering Task Force (IETF)*, 2015.

[JBMH15] Mike Jones, John Bradley, Maciej Machulak, and Phil Hunt. OAuth 2.0 Dynamic Client Registration Protocol (RFC7591). *Internet Engineering Task Force (IETF)*, 2015.

[JBS15] Mike Jones, John Bradley, and Nat Sakimura. JSON Web Token (JWT) (RFC7519). *Internet Engineering Task Force (IETF)*, 2015.

[JCBD18] M Jones, B Campbell, J Bradley, and William Denniss. OAuth 2.0 Token Binding-draft-ietf-oauth-token-binding-08. *Internet Engineering Task Force (IETF)*, 2018.

[JCL+17] Yunhan Jack Jia, Qi Alfred Chen, Yikai Lin, Chao Kong, and Z Morley Mao. Open doors for bob and mallory: Open port usage in android apps and security implications. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2017.

[JCM15] Mike Jones, Brian Campbell, and Chuck Mortimore. JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants (RFC7523). *Internet Engineering Task Force (IETF)*, 2015.

[JH12] Mike Jones and Dick Hardt. The OAuth 2.0 Authorization Framework: Bearer Token Usage (RFC6750). *Internet Engineering Task Force (IETF)*, 2012.

[JNC+20] Mike Jones, Anthony Nadalin, Brian Campbell, John Bradley, and Chuck Mortimore. OAuth 2.0 Token Exchange (RFC8693). *Internet Engineering Task Force (IETF)*, 2020.

[Jon15] Michael Jones. JSON Web Key (RFC7517). *Internet Engineering Task Force (IETF)*, 2015.

[Ker21] Brian Kerbs. Internet Bank Account Takeover of 1M Users. https://mrbriankrebs.medium.com/internet-bank-account-takeover-of-1m-users-without-user-interaction-4fc9141740a3, 2021 (last accessed January, 2021).

[Lan18] Patrik Lantz. DroidBox. https://github.com/pjlantz/droidbox, 2018.

[LBLF21] Torsten Lodderstedt, John Bradley, Andrey Labunets, and Daniel Fett. OAuth 2.0 Security Best Current Practice (draft-ietf-oauth-security-topics-18). *Internet Engineering Task Force (IETF)*, 2021.

[LC18a] Torsten Lodderstedt and Brian Campbell. Financial-grade API: JWT Secured Authorization Response Mode for OAuth 2.0 (JARM). *Internet Engineering Task Force (IETF)*, 2018.

[LC18b] Torsten Lodderstedt and Brian Campbell. Financial-grade API: JWT Secured Authorization Response Mode for OAuth 2.0 (JARM) (draft-ietf-oauth-jarm-2). *Internet Engineering Task Force (IETF)*, 2018.

[LF20] Torsten Lodderstedt and Daniel Fett. OpenID Connect for Identity Assurance 1.0. *The OpenID Foundation, specification*, 2020.

[Lin21a] Linkedin. Authenticating with OAuth 2.0. https://docs.microsoft.com/en-us/linkedin/, 2021 (last accessed February, 2021).

[Lin21b] LinkedinDev. Linkedin Android SDK. https://developer.linkedin.com/downloads, 2021 (last accessed February, 2021).

[LL09] Kelly D Lewis and James E Lewis. Web Single Sign-On Authentication using SAML. *International Journal of Computer Science Issues*, 2:41–48, 2009.

[LLW+12] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proceedings of the ACM SIGSAG conference on Computer and communications security, CCS 2012, Raleigh, NC, USA, October 16-18, 2012*, pages 229–240, 2012.

[LLWZ18] Xing Liu, Jiqiang Liu, Wei Wang, and Sencun Zhu. Android Single Sign-On Security: Issues, Taxonomy and Directions. *Future Generation Computer Systems*, 89:402–420, 2018.

[LM20] Wanpeng Li and Chris J Mitchell. User Access Privacy in OAuth 2.0 and OpenID Connect. In *EuroS&PW*. IEEE, 2020.

[LMC19] Wanpeng Li, Chris J Mitchell, and Thomas Chen. Oauthguard: Protecting user security and privacy with oauth 2.0 and openid connect. In *Proceedings of the 5th ACM Workshop on Security Standardisation Research Workshop*, pages 35–44, 2019.

[LMH13] Torsten Lodderstedt, Mark McGloin, and Phil Hunt. OAuth 2.0 Threat Model and Security Considerations (RFC6819). *Internet Engineering Task Force (IETF)*, 2013.

[LMSR21] Matteo Leonelli, Umberto Morelli, Giada Sciarretta, and Silvio Ranise. Secure pull printing with qr codes and national eid cards: A software-oriented design and an open-source implementation. In *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy*, pages 251–256, 2021.

[LWP+17] Fang Liu, Chun Wang, Andres Pico, Danfeng Yao, and Gang Wang. Measuring the Insecurity of Mobile Deep Links of Android. In *26th USENIX Security Symposium (USENIX Security 2017), Vancouver, BC, Canada, August 16-18, 2017*, pages 953–969, 2017.

[Mad15a] Paul Madsen. Mobile OS Developments and Native Application Authentication. Technical report, 2015.

[Mad15b] Paul Madsen. NAPPS has left the building (but is still on the front lawn). Technical report, 2015.

[Mic21a] Microsoft. Microsoft Identity Platform. `https://docs.microsoft.com/en-us/azure/active-directory/develop/v1-overview`, 2021 (last accessed February, 2021).

[Mic21b] MicrosoftDev. ADAL Android SDK. `https://github.com/AzureAD/azure-activedirectory-library-for-android`, 2021 (last accessed February, 2021).

[Mic21c] MicrosoftDev(V2). MSAL Android SDK. `https://github.com/AzureAD/microsoft-authentication-library-for-android`, 2021 (last accessed February, 2021).

[Mic21d] Microsoft(V2). Microsoft Identity Platform v.2. `https://docs.microsoft.com/en-us/azure/active-directory/develop/v2-overview`, 2021 (last accessed February, 2021).

[MMSW17] Christian Mainka, Vladislav Mladenov, Jörg Schwenk, and Tobias Wich. SoK: single sign-on security—an evaluation of openID connect. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 251–266. IEEE, 2017.

[MvOC21] Srivathsan G Morkonda, Paul C van Oorschot, and Sonia Chiasson. Exploring Privacy Implications in OAuth Deployments. *arXiv preprint arXiv:2103.02579*, 2021.

[NB19] Jorge Navas and Marta Beltrán. Understanding and Mitigating OpenID Connect Threats. *Computers & Security*, 84:1–16, 2019.

[NDP+17] Michael Nieles, Kelley Dempsey, Victoria Yan Pillitteri, et al. An introduction to information security. *NIST special publication*, 800:12, 2017.

[Now18] Nowsecure. Nowsecure. `https://www.nowsecure.com/solutions/mobile-app-security-testing/`, 2018.

[NS19] Mail.ru Nikita Stupin. Vulnerabilities of Mobile OAuth 2.0. `https://www.youtube.com/watch?v=vjCF_O6aZIg`, 2019. Insomni'hack Security Conference.

[NVI18] NVISO. NVISO. `https://www.nviso.be/`, 2018.

[NWA+17] Duc Cuong Nguyen, Dominik Wermke, Yasemin Acar, Michael Backes, Charles Weir, and Sascha Fahl. A Stitch in Time: Supporting Android Developers in Writing Secure Code. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security CCS 2017, Dallas, TX, USA, October 30-November 3, 2017*, pages 1065–1077, 2017.

[OAu19] OAuthWG. OAuth Service Provider. `https://oauth.net/code/`, 2019.

[OID16] OIDF. AppAuth Mobile Client SDK. `https://github.com/openid/AppAuth-Android`, 2016.

[OKT21a] OKTA. OKTA Authentication Guide. `https://developer.okta.com/quickstart/#/android/nodejs/express`, 2021 (last accessed February, 2021).

[OKT21b] OKTADev. OKTA SDK for Android. `https://github.com/okta/okta-oidc-android`, 2021 (last accessed February, 2021).

[OMGMR+10] Roberto Ortiz, Santiago Moral-García, Santiago Moral-Rubio, Belén Vela, Javier Garzás, and Eduardo Fernández-Medina. Applicability of security patterns. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 672–684. Springer, 2010.

[Ope18] Opensecurity. MobSF. `https://github.com/MobSF/Mobile-Security-Framework-MobSF`, 2018.

[Ope19] OpenID. Certified OpenID Connect Providers. https://www.openid.net/certification/, 2019.

[Phi21] Pieter Philippaerts. OAuch Security Best Practices and Threats Analyzer. https://oauch.io, 2021.

[Pin21a] PingIdentity. Mobile Application SSO Devloper Guide. https://www.pingidentity.com/developer/en/resources/napps-native-app-sso.html, 2021 (last accessed February, 2021).

[Pin21b] PingIdentityDev. PingOne Android SDK. https://github.com/pingidentity/pingone-customers-mobile-sdk/tree/master/Android, 2021 (last accessed February, 2021).

[PM12] Andreas Pashalidis and Chris J Mitchell. Privacy in identity and access management systems. In *Digital Identity and Access Management: Technologies and Frameworks*. IGI Global, 2012.

[Quo19] Quocirca. Global Print Security Landscape, 2019.

[RB11] Martin Rost and Kirsten Bock. Privacy by design and the new protection goals. *DuD*, 2009, 2011.

[RJ18] Justin Richer and Leif Johansson. Vector of Trust (RFC 8485). *IETF (Internet Engineering Task Force)*, 2018.

[RKJ06] Ronald S Ross, Stuart W Katzke, and L A Johnson. Minimum Security Requirements for Federal Information and Information Systems. *Federal Information Processing Standards Publication (NIST FIPS)*, 2006.

[RP09] Martin Rost and Andreas Pfitzmann. Datenschutz-schutzziele—revisited. *Datenschutz und Datensicherheit-DuD*, 33(6):353–358, 2009.

[Sak] Nat Sakimura. Authorization Delegation: A financial accounts aggregation use case. https://nat.sakimura.org/2016/01/29/authorization-delegation-a-financial-accounts-aggregation-use-case/. Accessed: 2021-03-25.

[SBA15] Nat Sakimura, John Bradley, and Naveen Agarwal. Proof Key for Code Exchange by OAuth Public Clients (RFC7636). *Internet Engineering Task Force (IETF)*, 2015.

[SBGM16] Alireza Sadeghi, Hamid Bagheri, Joshua Garcia, and Sam Malek. A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software. *IEEE Transactions on Software Engineering*, 43(6):492–530, 2016.

[SBJ+14] Nat Sakimura, John Bradley, Mike Jones, Breno De Medeiros, and Chuck Mortimore. OpenID Connect Core 1.0 incorporating errata set 1. *The OpenID Foundation, specification*, 335, 2014.

[SBJ21a] Nat Sakimura, John Bradley, and Edmund Jay. Financial-grade API - Part 1: Baseline. *Internet Engineering Task Force (IETF)*, 2021.

[SBJ21b] Nat Sakimura, John Bradley, and Edmund Jay. Financial-grade API - Part 2: Advanced. *Internet Engineering Task Force (IETF)*, 2021.

[SBJJ13] Nat Sakimura, John Bradley, Mike Jones, and E Jay. Openid connect dynamic client registration 1.0. *Internet Engineering Task Force (IETF)*, 2013.

[SCRS19] Amir Sharif, Roberto Carbone, Silvio Ranise, and Giada Sciarretta. A Wizard-based Approach for Secure Code Generation of Single Sign-On and Access Delegation Solutions for Mobile Native Apps. In *Proceedings of the 16th International Joint Conference on e-Business and Telecommunications, ICETE 2019 - Volume 2: SECRYPT, Prague, Czech Republic, July 26-28, 2019*, pages 268–275, 2019.

[SCSR20] Amir Sharif, Roberto Carbone, Giada Sciarretta, and Silvio Ranise. Automated and secure integration of the openid connect igov profile in mobile native applications. In *International Workshop on Emerging Technologies for Authorization and Authentication*, pages 50–70. Springer, 2020.

[SCSR21] Amir Sharif, Roberto Carbone, Giada Sciarretta, and Silvio Ranise. Best Current Practices for OAuth and OpenID Connect: A Study of their Adoption in Popular Providers and Top-Ranked Android Clients. *Under revision in Journal of Information Security and Application*, 2021.

[SH14] Hossain Shahriar and Hisham M Haddad. Content provider leakage vulnerability detection in android applications. In *Proceedings of the 7th International Conference on Security of Information and Networks*, pages 359–366, 2014.

[SMS11] Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. Rolecast: finding missing security checks when you do not know what checks are. In *Proceedings of the 26th ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2011, Portland, OR, USA, October, 2011*, volume 46, pages 1069–1084, 2011.

[SMS14] Alberto Siena, Mirko Morandini, and Angelo Susi. Modelling risks in open source software component selection. In *International Conference on Conceptual Modeling*, pages 335–348. Springer, 2014.

[STTPLB14] Guillermo Suarez-Tangil, Juan E Tapiador, Pedro Peris-Lopez, and Jorge Blasco. Dendroid: A text mining approach to analyzing and classifying code structures in android malware families. *Expert Systems with Applications*, 41(4):1104–1117, 2014.

[Sup18] SuperTeam. Super. `https://github.com/SUPERAndroidAnalyzer/super`, 2018.

[SWL19] Shangcheng Shi, Xianbo Wang, and Wing Cheong Lau. MoSSOT: An Automated Blackbox Tester for Single Sign-On Vulnerabilities in Mobile Applications. In *Proceedings of the ACM Asia Conference on Computer and Communications Security, Asia CCS 2019, Auckland, New Zealand, July 7-12, 2019*, pages 269–282, 2019.

[TRA18] TRACEDROID. TRACEDROID. `http://tracedroid.few.vu.nl/`, 2018.

[VAMC18] Luca Verderame, Alessandro Armando, Alessio Merlo, and Gabriele Costa. Approver. `https://www.talos-sec.com`, 2018.

[WSJ+19] K Wuyts, R Scandariato, W Joosen, M Deng, and B Preneel. LINDDUN: a privacy threat analysis framework, 2019.

[WW17] Rafael Weingärtner and Carla Merkle Westphall. A design towards personally identifiable information control and awareness in openid connect identity providers. In *CIT*. IEEE, 2017.

[WZL+15] Hui Wang, Yuanyuan Zhang, Juanru Li, Hui Liu, Wenbo Yang, Bodong Li, and Dawu Gu. Vulnerability Assessment of OAuth Implementations in Android Applications. In *Proceedings of the 31st annual computer security applications conference, ACSAC 2015, Los Angles, CA, USA, December 7-11, 2015*, pages 61–70, 2015.

[Yah21] Yahoo. Yahoo OAuth 2.0 Guide. `https://developer.yahoo.com/oauth2/guide/`, 2021 (last accessed February, 2021).

[YLS17] Ronghai Yang, Wing Cheong Lau, and Shangcheng Shi. Breaking and Fixing Mobile App Authentication with OAuth2.0-based Protocols. In *Proceedings of the 15th International Conference on Applied Cryptography and Network Security, ACNS 2017 - Volume 10355:LNCS, Kanazawa, Japan, July 10-12, 2017*, pages 313–335, 2017.

[ZE14] Yuchen Zhou and David Evans. SSOScan: Automated testing of web applications for Single Sign-On vulnerabilities. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 495–510, 2014.

[Zha18a] Yuan Zhang. Detecting Third-Party Libraries in Android Applications with High Precision and Recall. https://github.com/yuanxzhang/LibPecker, 2018.

[Zha18b] Yury Zhauniarovich. StaDynA. https://github.com/zyrikby/StaDynA, 2018.

[ZKS+20] Zhiyi Zhang, Michał Król, Alberto Sonnino, Lixia Zhang, and Etienne Rivière. EL PASSO: Privacy-preserving, Asynchronous Single Sign-On. *arXiv preprint arXiv:2002.10289*, 2020.

[ZZL17] Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. Authscope: Towards Automatic Discovery of Vulnerable Authorizations in Online Services. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30-November 3, 2017*, pages 799–813, 2017.

Appendix A

Downloaded Apps and Automatic Selection S1.

A.1 Details

By using the PlaystoreDownloader tool [Cla17], we download 2505 top-ranked apps. The total number of apps for each category is reported in Table A.1 (column "# Downloaded Apps").

Table A.1: Details about Downloaded Apps and Automatic Selection S1.

App Category	# Downloaded Apps	Amazon	Auth0	Box	BeeMinder	Dropbox	Facebook	Google	HiDrive	IBM	Linkedin	Microsoft	OKTA	PING	Yahoo	Tot. Detected Insts	Tot. Detected Apps
Business	192	5	0	7	0	7	50	5	0	0	4	9	0	0	2	89	63
Communication	189	0	0	1	0	0	47	1	0	0	1	1	0	0	6	57	50
Dating	186	1	0	1	0	0	113	0	0	0	0	0	0	0	0	115	115
Education	194	0	0	2	0	0	57	2	0	0	0	0	0	0	0	61	58
Health & fitness	198	0	0	0	0	0	41	2	0	0	2	0	0	0	0	45	43
Medical	195	1	0	1	0	1	41	0	0	0	0	0	0	0	0	44	43
Maps & Nav	197	2	0	1	0	0	60	1	0	0	0	0	0	0	1	65	63
Personalization	188	1	0	0	0	0	50	0	0	0	1	1	0	0	0	53	51
Productivity	195	3	0	6	0	9	56	6	0	0	1	9	0	0	2	92	63
Shopping	194	6	0	3	0	0	124	2	0	0	1	0	0	0	2	138	129
Social	196	2	0	1	0	0	106	2	0	0	0	0	0	0	0	111	107
Tools	186	1	0	0	0	0	45	1	0	0	0	1	0	0	0	48	46
Travel	195	3	0	2	0	0	89	0	0	0	0	0	0	0	0	94	90
Total	2505	25	0	25	0	17	879	22	0	0	10	21	0	0	13	**1012**	921

A.2 Accuracy

In our App Automatic Selection S1. we are not interested in ensuring "accuracy", namely in completely avoiding false negatives (FNs) and false positives (FPs). We only aim at selecting a set of apps that integrate (at least) one of the IdMPs of Table 4.1. Therefore, for clarity, in this section we point out some limitations of our selection procedure, which do not impact our analysis, but could be helpful for interested readers.

A.2.1 False Negative *C* Apps in Download and S1.

The selection procedure (download and S1.) we used is not accurate in detecting all the apps that use the selected IdMPs, namely we have false negatives. In particular, we point out the following reasons:

- In Table A.1 the numbers of downloaded apps for each category are slightly different, because PlaystoreDownloader failed to download some apps, due to a known issue.[1]
- Our python script used in S1. looks for a string identifying the most recent endpoints at the time of our analysis. Thus, of course, our script cannot detect *C* apps which are using different (deprecated) endpoints.
- Our python script used in S1. looks for the endpoints of IdMPs in the code of a downloaded *C* app. Thus, it is not able to detect *C* apps which are using Dynamic Code Loading (DCL) to dynamically include the endpoints.

A.2.2 False Positive *C* Apps in S1 (detected in S2.)

As mentioned in Section 4.3.1, some of the apps selected in S1. are false positives. Based on our observation by manually inspecting the code in S2., we have identified two main reasons for FPs.

- The *C* app integrates the IdMP SDK, but the specific OAuth/OIDC functions are not invoked within *C* app. As an example consider the scenario where the native app developer integrates both Facebook identity and advertisement SDKs but the *C* app invokes only the Facebook Ads functionality.
- The *C* app contains the IdMPs endpoint, but it is using traditional username/password to perform login. Google is a good example for this case, as it provides a simple API for

[1] https://github.com/ClaudiuGeorgiu/PlaystoreDownloader/issues/18

developers to perform user authentication called Google Sign-In[2]. Indeed, this method simplify the user authentication by using the Google account already configured within the user's device.

Finally, for interested readers, the companion website provides a table that summarizes FPs with regard to the aforementioned two main reasons for each IdMP.

[2] https://developers.google.com/identity/

Appendix B

Static Analysis Tools

Table B.1 reports the open-source available automatic source code analysis tools. Based on the observation, that we had on the capabilities of the automatic analysis tools, we select SUPER tools to support us in the app analysis phase. In addition, we use JADX in our manual analysis, which is a known dex to java decompiler.

B.1 JADX

JADX[1] provides a GUI interface that allows one to load the app *APK* file. Then, it decompiles the *APK* into its original data structure.

We use this tool to manually navigate the App source code in order to check the scenario (SSO or AD), the SDK support (Official or unofficial/no SDK), BCPs compliancy (BCP_{flow}, BCP_{UA}, and *PKCE* usage in case of Custom URI redirection (BCP_{PKCE})), and implementation issues (the `client_secret` in plaintext, and HTTPS configuration). In particular, we start from the *AndroidManifest.XML* file to find SSO and/or AD related interested activities. By knowing the names, we navigate to related classes to check the SSO and/or AD solution implementations. Alternatively, a simple way to avoid checking the manifest to find the name of interested SSO/AD activities is to use the search feature of the tool (if available, as some times JADX fails to index all the files within the decompiled files) to look for the IdMPs endpoint. Thus, it leads into a list of interested activities, which integrates SSO and/or AD solution implementations. After that, we perform the following steps: (i) check the activity code, its implemented interface(s) (if any), and included function(s) to check for *UA* type, *PKCE* implementation and the `client_secret` usage in plaintext; (ii) if the `client_secret` is not declared directly within the SSO and/or AD activity and the search feature is available, we look for possible strings within the decompiled files; and

[1] `https://github.com/skylot/jadx`

Table B.1: Static Analysis Tools Overview.

Tool	HTTP Traffic	WebView Detect	String Detect	Malicious Behavior/Malware	Data Leaks	App Clone	Inter-App Communication/Intent Vuln	Permision Misuse	Crypto Fault	Code Verification	3rd-Library Vuln
Amandroid	☐	☐	☐	☐	☑	☐	☑	☐	☑	☐	☐
Androguard2	☐	☐	☐	☐	☐	☑	☑	☐	☐	☐	☐
Android-app-analysis	☐	☐	☐	☐	☐	☐	☐	☑	☐	☐	☐
Cassandra	☐	☐	☐	☐	☑	☐	☐	☐	☐	☐	☐
Dflow + DroidInfer	☐	☐	☐	☐	☑	☐	☐	☐	☐	☐	☐
DidFail	☐	☐	☐	☐	☑	☐	☐	☐	☐	☐	☐
FlowDroid	☐	☐	☐	☐	☑	☐	☐	☐	☐	☐	☐
Concurrency	☑	☐	☐	☐	☐	☐	☐	☐	☑	☐	☐
PScout	☐	☐	☐	☐	☐	☐	☐	☑	☐	☐	☐
Xiao et al.	☐	☐	☐	☐	☑	☐	☐	☐	☐	☐	☐
CooperDroid	☐	☐	☐	☑	☑	☐	☐	☐	☐	☐	☐
TaintDroid	☐	☐	☐	☐	☑	☐	☐	☐	☐	☐	☐
DroidScope (DECAF)	☐	☐	☐	☑	☑	☐	☐	☐	☐	☐	☐
AndroWarn	☐	☐	☐	☑	☐	☐	☐	☐	☐	☐	☐
SPARTA	☐	☐	☐	☑	☐	☐	☐	☐	☐	☐	☐
DroidRA	☐	☐	☐	☑	☐	☐	☐	☐	☐	☐	☐
SUPER	☑	☑	☐	☐	☐	☐	☐	☐	☑	☑	☐
ClassyShark	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
StaCoAn	☐	☐	☑	☐	☐	☐	☐	☐	☐	☐	☐
JAADS	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
QARK	☑	☑	☐	☐	☑	☐	☑	☐	☑	☑	☐
TriFlow	☐	☐	☐	☑	☐	☐	☐	☐	☐	☐	☐
LibID	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☑
LibRadar	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☑
LibD	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☑
LibPecker	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☑
MobSF	☐	☐	☑	☑	☑	☐	☐	☑	☑	☑	☑

(*iii*) in the case of HTTPS redirection within the app, we check for the proper configuration of App Link within the *AndroidManifest.XML*.

B.2 SUPER Android Analyzer

SUPER[2] is an automatic static analysis tool for android apps. It provides an informative analysis report that is useful to guide the source code analysis. We use the tool for detection of the Web View within apps, in case of strongly-obfuscated apps (i.e. not only the function names but also the variables are obfuscated). In order to perform the app analysis using SUPER Android Analyzer, we perform the following steps: (*i*) configure the tool in our working environment; (*ii*) download the *APK* file of the app under analysis; (*iii*) run the tool; and (*iv*) analyze the generated report by navigating through the identified issues. In particular, we check for the embedded browser related issues in the app. Then, we check the path within the app to assure that our finding is correlated to the SSO and/or AD solution implementations.

[2] `https://github.com/SUPERAndroidAnalyzer/super`

Appendix C

Dynamic Analysis Tools

This section presents some of the Android dyanmic analysis tools that we find them more efficient in the sense of security analysis. As some of these tools are open-source and some of them are not, we decide to categorize them into two main classes: (i) non-commercial, and (ii) commercial tools. In the following, we briefly describe the capabilities of each tool that we select for both commercial and non-commercial categories.

C.1 Non-Commercial Dynamic Analysis Tools

This section summarizes non-commercial dynamic analysis tools which are accessible online.

C.1.1 DROZER

DROZER [FS18] is a real-time tool that allows performing dynamic analysis by interacting with the app. DROZER composes of an agent installed on a smartphone and a desktop application to interact with the agent. The tool provides helpful information such as the app's permission, exported activities, broadcast receivers, and content providers, which can be used to find possible security issues.

C.1.2 DroidBox

DroidBox [Lan18] is a dynamic analysis tool, which monitors the app behavior during running on an emulator or test device. DroidBox extracts information concerning the app behavior, such as intents, network traffic, string saved on shared preferences, SMS sent, etc. The tool generates

a report that contains the data sent by the app, the started intents, and the information wrote in the shared preference.

C.1.3 Inspeckage

Inspeckage [ap18] is an Xposed module that enables security app pentester to analyze the app during its running on an emulator dynamically. The tool provides helpful information like package information, the content of shared preference, crypto APIs used by the app, SQLite queries, intercepted traffic, files accessed by the app, and actions performed in the Web View (if the app uses it).

C.1.4 CuckooDroid

CuckooDroid [Cuc18] performs both static and dynamic analysis for android apps. The tool generates a comprehensive report concerning app behavior like executed shell commands, queried accounts, dynamically loaded files, SMS messages sent and received, registered broadcast receivers, etc. In addition, CuckooDroid provides a list of possible vulnerabilities for the app under analysis without any additional details about the reason and how to mitigate the vulnerabilities.

C.1.5 StaDynA

StaDynA [Zha18b] is an open-source static and dynamic analysis tool for android apps. The tools consist of two parts: a server and a client. While the server-side is responsible for interaction with the static analysis tool, the client code runs on either the emulator or test device to perform dynamic analysis. The tool provides a method call graph by using the static analysis. It then uses the provided method call graph during dynamic analysis to analyze the code added through dynamic code loading and reflection APIs. Finally, it generates a report regarding the tool's findings during static and dynamic analysis.

C.1.6 MobSF

MobSF [Ope18] is an automated pentesting, malware analysis, and security framework for apps developed for different platforms (Android, iOS, Windows), and it is capable of performing static and dynamic analysis. The tool can be installed easily by using the docker file provided in the tool GitHub repository [Ope18]. After installation, you need to run the MobSF server and upload the APK file of your android app, and then it starts the static and dynamic analysis on the app. Finally, it generates a report using the collected data during the analysis which contains

information like malicious app behavior, detected strings as possible API secrets, permission misuse, outdated/faulty crypto APIs, etc.

C.2 Commercial Dynamic Analysis Tools

While the previous section represents the non-commercial dynamic analysis tools which are open-source, this section presents some of the commercial tools that are used in the wild.

C.2.1 AppKnox

AppKnox [App18b] is a commercial tool that allows performing automatic dynamic analysis of Android apps. In order to perform an analysis with Appknox, developers need to create an account within their website. The tool provides a comprehensive report that contains the recognized vulnerabilities, a brief description of the vulnerability, the security level, and its business implication. Note that it checks the app against the vulnerabilities reported in OWASP mobile top 10 document [1].

C.2.2 AppCritique

AppCritique [App18a] provides both free and paid plans. While the former only provides a report with possible vulnerabilities within developers app, in the latter, developers can keep in touch with their security experts to get help on how to fix the founded issues within your app. The generated report by the tool contains information like package information, certificate information, used third-party libraries, request permissions by the app, and a list of recognized vulnerabilities. It is worth mentioning that the vulnerabilities list contains the name, a description of the issue. It sometimes highlights the part of the app code that is the origin of the vulnerability.

C.2.3 IBM Application Security

IBM Application Security [IBM18] is an automated static/dynamic analysis tool. The tool is accessible through a reserved area for both free and paid customers. Customers upload their app APK file through the tool website. The tool generates a comprehensive report containing a list of issues based on the OWASP Top 10 Mobile Security Project. The report sorts the founded implementation issues based on their severity, provides an overview for each issue, and recommends a possible way to mitigate it.

[1] `https://www.owasp.org/index.php/Category:OWASP`

C.2.4 ImmuniWeb

ImmuniWeb [Imm18] Mobile provides an automated dynamic analysis tool, which checks the app against vulnerabilities reported in the OWASP Top 10 Mobile Security Project. The final report provides a list of founded vulnerabilities, a brief description for each vulnerability, and a recommendation on fixing the reported problems. In their paid subscription, developers can ask for a manual analysis of their apps which an app security expert performs.

C.2.5 NVISO

NVISO [NVI18] is an automatic static/dynamic analysis tool. Developers need to upload their app APK file to the NVISO website to start the analysis procedure and get informed when the analysis is finished. The final report is divided into static and dynamic analysis sections. The static analysis part provides information about requested permissions by the app, the hardcoded strings, and etc. Also, it checks the app against known malware by using external services (e.g., Virus Total). The dynamic analysis part provides details of files accessed by the app, network activities, SMS sent and received, crypto APIs used by the app, etc.

C.2.6 TraceDroid

TraceDroid [TRA18] provides dynamic analysis methods that enable developers to analyze the app at runtime. To analyze the app, developers need to provide their app APK file to TraceDroid website and it takes a few minutes to complete the analysis. The final report contains netwrok activities, screenshot of the app activities, and Message Call Graph.

C.2.7 AppWatch

Appwatch [App18c] is an automatic static and dynamic analysis tool for apps that provides in-depth app security analysis. The tool static analysis module can detect hardcoded sensitive data, Web View-based vulnerabilities, and code implementation flaws. Appwatch dynamic analysis module analyzes the app behavior to find network-based vulnerabilities, insecure inter-process communication problems, insecure data storage, etc. The tool provides a comprehensive report w.r.t. the founded issues and a brief explanation and how to mitigate them.

C.2.8 Nowsecure

Nowsecure [Now18] provides security and privacy testing for apps. Nowsecure provides compliance requirements against a wide range of regulations such as OWASP, GDPR, HIPAA, and FFIEC (to name just a few of them). It performs both static and dynamic analysis to find possible security and privacy issues like improper SSL implementation, data leakage, insecure credential transmission, hardcoded credentials, etc.

C.2.9 App-Ray

App-Ray [AR18] is a fully automated static and dynamic app analysis tool. To analyze the app, native app developers can submit their app to the App-Ray website and get informed when the app analysis has been finished. The tool consists of three main modules: Metadata analysis, static analysis, and dynamic analysis. The Metadata analysis module provides basic information about the app like native and third-party libraries, activities, services, and content providers used in the app under analysis. In addition, it uses external services (Virus Total) to check the app against known Malware. The static analysis module checks the apps against malicious or vulnerable code patterns. Also, it checks the app TrustManager interface implementation to check its correct implementation as implementation flaws may cause man-in-the-middle attack. Finally, the dynamic analysis module runs the app in an emulated environment to analyze the app behavior, monitor network traffic, and detect possible data leakage, user tracking, etc.

C.2.10 Approver

Approver [VAMC18] is a fully automated static and dynamic app analysis tool that detects, evaluates, and provides comprehensive reports on the security risks posed by the app. The tool checks the app against 60 different vulnerability patterns, looks for permission misuse, malicious code, policy violation patterns concerning OWASP Top 10 Mobile Security Risks, SSL implementation flaws, hard-coded credentials, crypto API misuse, privacy leaks, etc. The tool creates a comprehensive analysis report that contains risk scoring, a brief explanation for the founded problems, and describes how to mitigate the founded issues within the app.

Appendix D

Popular OAuth/OIDC IdMPs Analysis Result (2019)

As mentioned in Section 4.2, we perform a preliminary IdMPs analysis in October 2019, and we report the compliance analysis results in Table D.1. In the following, we summarize our findings with regard to the research questions defined in Section 4.2.3.

D.1 Discussion on *R1*.

Regarding the flow (4th column in Table D.1), even if the *Authorization Code* flow with *PKCE* is the recommended flow (BCP_{flow}), IBM is also supporting the *RO Password Credential* flow and 4 IdMPs (Auth0, Ping, Beeminder and Amazon) are still permitting the usage of Implicit flow, which is not recommended to use within native apps as it cannot be protected by the *PKCE*. Note that, while the *Authorization Code* flow is actually the default flow for both Ping and Auth0 SDKs, there is still the possibility for a native app developer to choose the Implicit flow. In addition, the Ping SDK API reference [Pin21a] suggests the usage of the *Implicit* flow as the default grant type, because it enables inexperienced developers to effortlessly learn and integrate OAuth/OIDC solutions within their native apps. Therefore, the Ping suggestion to use *Implicit* flow for native apps may mislead native app developers who intend to manually integrate the OAuth/OIDC solutions. Indeed, native app developers may unintentionally integrate *Implicit* flow within their native apps, even if this is strongly discouraged by BCPs [DB17]. In case of Amazon, the *Implicit* flow is secured by two alternative security measures. First of all, Amazon requires to store the app package name and app developer certificate to create an API Key, which is used to identify the *C* app. Secondly, Amazon highly recommends the usage of the *tokenInfo* endpoint to avoid *RO* impersonation in the *Implicit* flow by verifying the authenticity of the access token through a secure HTTP call to the *tokenInfo* endpoint. Facebook supports a modified

Table D.1: IdMPs Status 2019.

IdMP Name	Scenario		SDK Support		
	SSO	AD	Flows	UA	PKCE
Google [19]	●	●	code w/o secret	CT	✓
Facebook [15, 16]	◗	●	*modified implicit**	CT, *N**	N/A
Yahoo [53]	●	●	-	-	-
Amazon [2, 3]	◗	●	code w/o secret, *implicit**	Ex	✓, N/A
Linkedin [26, 27]	◗	●	code *w/ secret*	Ex	x
Microsoft v1 [33, 34]	●	●	code w/o secret	*Em*	*x**
Microsoft v2 [36, 35]	●	●	code w/o secret	CT	✓
DropBox [13, 14]	❍	●	code *w/ secret*	Ex	*x**
OKTA [41, 42]	●	◗	code w/o secret	CT	✓
Box [7, 8]	❍	●	code *w/ secret*	*Em*, *N**	x, *x**
Auth0 [4, 5]	●	◗	code w/o secret, *implicit*	CT	✓, N/A
HiDrive [21, 22]	❍	●	*modified code w/ secret*	Ex	*x**
Beeminder [6]	❍	●	-	-	-
Ping [44, 45]	●	◗	code w/o secret, *implicit*	CT	✓‡
IBM [23, 24]	●	❍	code w/o secret, *pass*	*N**	✓‡

Scenario	●	IdMP supports OIDC or OAuth
	❍	IdMP does not support OIDC or OAuth
	◗	IdMP supports a SSO Login different from OIDC or supports OAuth to access developer resources but does not expose its APIs
Flows	code	*Authorization Code* flow with (w/) or without (w/o) the use of `client_secret`
	implicit	*Implicit* flow
	pass	*RO Password Credential* flow
UA	Ex	External browser
	Em	Embedded browser
	CT	External browser supporting Custom Tab
	N	Native app
PKCE	✓	The SDK supports the *PKCE* protocol
	✓‡	The SDK supports *PKCE* but it is optional
	x	The SDK does not support *PKCE*
	N/A	*PKCE* cannot be used as the flow is not the *Authorization Code* flow

Implicit flow by requiring the interaction with their apps together with the storing of the app developer package name and app developer certificate hash to verify the identity of the *C* app. In addition, Facebook enforces further security checks that can be activated from the app developer console, such as strict mode (avoid redirect hijacking by requiring exact match from OAuth redirect list), enforce HTTPS scheme registration, disable embedded OAuth login, app review process (in case the *C* app needs to access APIs and features that are not allowed by default, such as page public content access), and so on.

Regarding *PKCE* (6th column in Table D.1), we discovered that among the 13 IdMPs that have a client SDK (*i*) 5 support *PKCE*, (*ii*) 2 optionally support *PKCE* (in case of Ping the *PKCE*

query parameters are not automatically sent in the request as the app developer is required to set an optional property in the SDK; while for IBM the app developer can disable the *PKCE* support from the console), (*iii*) 2 do not support *PKCE* (Linkedin and Box) and violate BCP_{PKCE} , (*iv*) Facebook does not support *PKCE* due to the fact of using another grant flow rather than the traditional *Authorization Code*, and (*v*) 3 (Microsoft v1, HiDrive, DropBox) do not support *PKCE* as they use different security measurement to protect the code interception. While Microsoft v1 uses the package name and certificate to identify the *C* app, HiDrive prompts the user to copy a code and paste it into the app to obtain the Authorization code and, finally, DropBox raises a security alert if another app registers the same Custom URI scheme as the legitimate app.

Another problem in Microsoft and Box SDK (5th column in Table D.1) is the usage of an embedded browser (against BCP_{UA}).

D.2 Discussion on *R2*.

Our analysis proves that 7 IdMPs (Facebook, Beeminder, Box, Dropbox, Hidrive, Linkedin, Yahoo) are not compatible by the default configuration of AppAuth. For Facebook, Beeminder and HiDrive is due to the different grant types, in the case of Box, Dropbox, and Yahoo due to the need to add `client_secret` and to remove *PKCE*, and for Linkedin due to the different `access_token` response format, which is missing the `token_type` field. The `token_type` field is required as mentioned in the OAuth Authorization Framework [Har12]. 7 IdMPs (Google, Amazon, Microsoft v2, OKTA, Auth0, Ping, IBM) are compatible with the default setting of AppAuth as they are following all the BCPs recommendations for native apps .

Appendix E

Top-Ranked Google Play Store Compliance Analysis Results (2019)

This section aims to summarize our top-ranked Google Play Store App Analysis results performed in 2019. Table E.1 represents the selection procedures to reach the final resulting data set (RD2019).

Note that we considered the same selection criteria to analyze the OAuth/OIDC implementations within the top-ranked apps; we only summarize our finding concerning the research questions defined in Section 4.3.4.

E.1 Discussion on *R1*

In Table E.2, we report the results per IdMP by specifying BCPs coverage of each app instance belonging to our Resulting Dataset, where: with *x*/3 we mean that the app instance is following *x* of the aforementioned recommendations. We call *fully compliant* with BCPs only app instances with a score of 3/3. From the results, we have that:

- 4/87 app instances are *fully compliant*: 2 for Amazon and 2 for Microsoft;
- all the app instances for Box, Dropbox, Facebook, Google, Linkedin and Yahoo are *not fully compliant* with BCPs.

Finally, referring to the number of *C* apps (instead of the app instances), we have that 57/60 (95%) of them are *not fully compliant* with BCPs.

Table E.1: Automatic and Manual selection (S1.-S2.), Static Analysis (S3.), Reaching 10 Apps (S4.) and Resulting Datasets (RD 2019, RD 2021).

IdMP	S1.	S2.		S3.				S4.	RD 2019
	# Detected Apps	FPs	Selected in S2 (i.e. #Selected S1−FPs)	Code not avail.	Strongly obfuscated	Premium not avail.	Selected in S3 (i.e. Selected S2−not avail.)	# Similar Apps	# Final Apps 2019
Amazon	25	22	3	0	0	0	3	7	10
Auth0	0	-	-	-	-	-	-	-	0
Box	25	13	12	0	0	1	11	-	11
BeeMinder	0	-	-	-	-	-	-	-	0
DropBox	17	0	17	1	0	1	15	-	15
Facebook	879	11*	10*	0	0	0	10	-	10
Google	22	12	10	2	1	2	7	3	10
HiDrive	0	-	-	-	-	-	-	-	0
IBM	0	-	-	-	-	-	-	-	0
Linkedin	10	4	6	1	0	0	5	5	10
Microsoft	21	9	12	1	0	0	11	-	11
OKTA	0	-	-	-	-	-	-	-	0
PING	0	-	-	-	-	-	-	-	0
Yahoo	13	4	9	0	0	0	9	1	10
Total Instances	**1012**	**77**	**79**	**5**	**1**	**4**	**71**	**16**	**87**
Total Apps	**921**	**72**	**49**	**5**	**1**	**2**	**46**	**16**	**60**

E.2 Discussion on *R2*

In our analysis, 38/87 app instances use the official SDK. Out of these, only 4/38 (2 for Amazon and 2 for Microsoft) are *fully compliant* with BCPs.

For the remaining 34 app instances *not fully compliant*, to understand if the violation of BCPs is due to the SDK officially provided by the IdMP or is introduced by the *C* developer, we have compared the features gathered from this analysis with Table 4.1. In addition, by considering the possible threats from Table 3.3, we have that:

- 5 app instances (4 Amazon, 1 Microsoft) are *not fully compliant* due to developer choices: they remove the support to *PKCE* which is mandatory for native apps. In the case there are not any additional security mechanisms, app instances would be vulnerable to token

Table E.2: Security BCPs coverage per IdMP.

IdMP	Sec. BCPs Coverage 0/3	1/3	2/3	3/3	% of app instances not fully compliant
Amazon	1	0	7	2	80%
Box	11	0	0	0	100%
Dropbox	6	9	0	0	100%
Facebook	10	0	0	0	100%
Google	4	2	4	0	100%
Linkedin	9	1	0	0	100%
Microsoft	2	6	1	2	82%
Yahoo	9	1	0	0	100%
Total Instances	52	19	12	4	95% (83/87)

leakage (T_{token}).

- 17 app instances (6 Box, 8 Dropbox, 3 Microsoft) are *not fully compliant* due to the IdMP. All of them do not support *PKCE*, which may lead to T_{token} in lack of additional security mechanisms. 14 app instances (6 Box, 8 Dropbox) support the *Authorization Code* flow with secret that may lead to client impersonation (T_{secret}) if the `client_secret` value is stored in plaintext within the *C* app. Finally, Microsoft supports an embedded browser that is highly discouraged within third-party apps, as its usage can lead to leak of user credentials (T_{cred}).
- 12 app instances (10 Facebook, 2 Dropbox) are *not fully compliant* due to both IdMP and developers. The developer fault is using an embedded browser instead of external that could make app instances susceptible to T_{cred}. While the IdMP violation is related to the flow (*Implicit* for Facebook and use of the `client_secret` for Dropbox). While in the case of Facebook the usage of *Implicit* flow is not a problem as they implement additional security mechanisms to avoid T_{token}, in case of Dropbox, the usage of the `client_secret` (in plaintext) may lead to T_{secret}.

In addition, we can observe that:

- $\sim$ 76% (26/34) are using a flow different from the *Authorization Code* flow without secret (10 *Implicit*, 17 *Authorization Code* with secret).
- $\sim$ 62% (21/34) are using an embedded browser.
- $\sim$ 71% (24/34) are omitting *PKCE*.

E.3 Discussion on *R3*

In our analysis, 49/87 app instances use the unofficial or no SDK, which are all *not fully compliant*. In particular, we noticed:

- $\sim$ 76% (37/49) are using a flow different from *Authorization Code* without secret.
- $\sim$ 82% (40/49) are using an embedded browser.
- $\sim$ 96% (47/49) are omitting *PKCE*.

E.4 Discussion on *R4*

In this research question, we report developers wrong implementation choices and in particular, we check the HTTPS scheme configuration and the storing of the `client_secret`.

Our analysis results is as the following:

- only 3 app instances are using the HTTPS scheme; out of these, 2 do not specify the field "`autoVerify=TRUE`" in the HTTPS scheme declaration; the lack of this field results in a wrong verification process of the app identity and in the absence of additional security mechanisms would lead to T_{code} and then T_{token}.
- 52 app instances are sending the `client_secret`; out of these, 26 store the secret in plaintext within apps that makes C apps susceptible to T_{secret}.

The problem of using `client_secret` in a native app is due to its nature. By definition, being a public client, a native app is not able to keep confidential any value. So OAuth/OIDC solutions should not base the client authentication on this value, especially if it is stored in plaintext. However, as we have previously shown many IdMPs are still requiring the sending of the secret. If the secret is necessary for backward compatibility, in order not to compromise the web scenario, a good practice is to use different `client_secret` value for the native scenario. For the 52 app instances sending the `client_secret`, we have that only 16 app instances are using different `client_secret` values for the web scenario (4 Google, 2 Microsoft, 10 Yahoo).