View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Archivio della ricerca - Fondazione Bruno Kessler

A Wizard-based Approach for Secure Code Generation of Single Sign-On and Access Delegation Solutions for Mobile Native Apps

Amir Sharif[1,2][a], Roberto Carbone[1][b], Silvio Ranise[1][c] and Giada Sciarretta[1][d]
[1] *Fondazione Bruno Kessler, Trento, Italy*
[2] *DIBRIS- University of Genoa, Genoa, Italy*

Keywords: Single Sign-On, OAuth 2.0, Android Security, Identity Management, Android Native Applications.

Abstract: Many available mobile applications (apps) have poorly implemented Single Sign-On and Access Delegation solutions leading to serious security issues. This could be caused by inexperienced developers who prioritize the implementation of core functionalities and/or misunderstand security critical parts. The situation is even worse in complex API scenarios where the app interacts with several providers. To address these problems, we propose a novel wizard-based approach that guides developers to integrate multiple third-party Identity Management (IdM) providers in their apps, by *(i)* "enforcing" the usage of best practices for native apps, *(ii)* avoiding the need to download several SDKs and understanding their online documentations (a list of known IdM providers with their configuration information is embedded within our approach), and *(iii)* automatically generating the code to enable the communication with the different IdM providers. The effectiveness of the proposed approach has been assessed by implementing an Android Studio plugin and using it to integrate several IdM providers, such as OKTA, Auth0, Microsoft, and Google.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

Identity Management (IdM) solutions are the main enabler for the development of applications that use the functionalities offered by Application Programming Interface (API) platforms. A widely adopted approach to integrate IdM solutions in mobile applications (hereafter, apps)—with the goal of striking the best possible trade-off between security and usability—is to use API for IdM made available from third-party IdM providers (hereafter IdMPs) such as Google, OKTA, and Auth0 (to name but a few). The following two are the scenarios in which IdMPs are most frequently used:

Single Sign-On (SSO) Login. solutions are widely used by app developers to authenticate the user leveraging an IdMP, which allow users for using the same credentials along different apps.

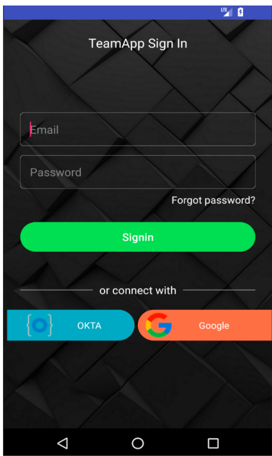
Access Delegation. allows users to delegate to the app the access to an API trough an IdMP.

[a] https://orcid.org/0000-0001-6290-3588
[b] https://orcid.org/0000-0003-2853-4269
[c] https://orcid.org/0000-0001-7269-9285
[d] https://orcid.org/0000-0001-7567-4526

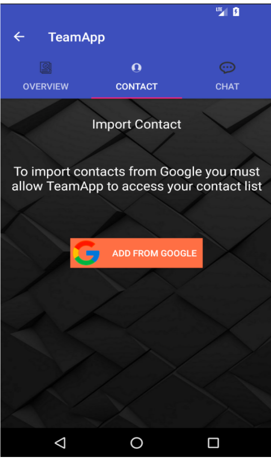
OAuth 2.0 (Hardt, 2012) (hereafter OAuth) and OpenID Connect (Sakimura et al., 2014) (hereafter OIDC) are among the most popular open standards for access delegation and SSO Login, respectively. Recently, the OpenID Foundation made an effort to support app developers by releasing AppAuth, a client SDK for native apps to authenticate and authorize end-users by communicating with OAuth and OIDC providers, beside implementing the security and usability best practices.

In this work, we analyze the compliance of popular IdMPs with the OAuth/OIDC best current practices for native apps. Our analysis reveals that, unfortunately, many IdMPs still do not follow the best practices. In addition, many apps have poorly implemented IdM mechanisms because of inexperienced, distracted, or overwhelmed developers (Liu et al., 2018). In general, developers with little security awareness, prioritize code functionality and ignore the security implications of their choices. Although IdMPs try to mitigate security issues by providing documentations, if developers are not security experts and do not understand security critical code, this may lead to serious security vulnerabilities. According to (Yang et al., 2017), 41% of 600 top-ranked US and Chinese Android apps that use Facebook, Sina, and

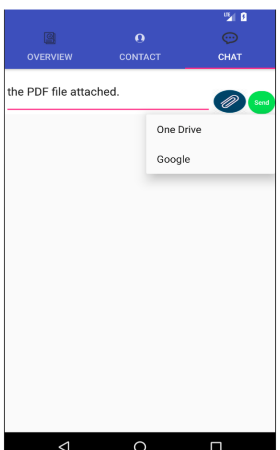
Sharif, A., Carbone, R., Ranise, S. and Sciarretta, G.
A Wizard-based Approach for Secure Code Generation of Single Sign-On and Access Delegation Solutions for Mobile Native Apps.
DOI: 10.5220/0007930502680275
In *Proceedings of the 16th International Joint Conference on e-Business and Telecommunications (ICETE 2019)*, pages 268-275
ISBN: 978-989-758-378-0
Copyright © 2019 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved



(a) Login Activity.



(b) Contact Activity.



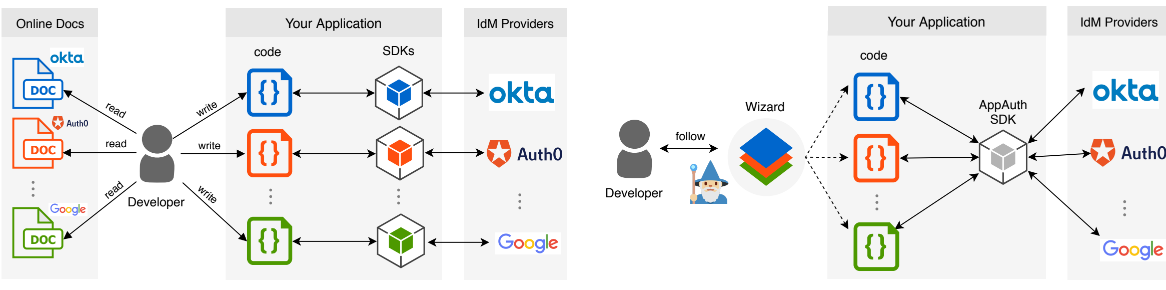
(c) Chart Activity.

Figure 1: TeamApp screen shots.

Google as IdMP providers are vulnerable. The situation is even worse in complex scenarios—which are more and more frequent with the advent of the API economy—where the app interacts with many IdMPs. To illustrate, let us describe a typical use case scenario in which the mobile app *TeamApp* helps team members to improve collaboration and the sharing of information. 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 1a). 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 1b) 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 1c). The effort required to implement *TeamApp* is substantial. As shown in Figure 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; and (**P3**) write different pieces of code to integrate the SDKs. Furthermore, in such a kind of scenarios where multiple IdMPs must be integrated, AppAuth cannot be easily adopted, because the available documentation is tailored to interact with a single IdMP. Therefore, app developers have two options: either reuse the same classes and configuration files of AppAuth with different names in order to avoid conflicts, or should implement some additional methods from scratch. Both options are not only time-consuming but also error-prone, especially for inexperienced developers.

To overcome these difficulties, we present a novel approach based on a wizard to support developers in the integration of multiple IdMPs for SSO Login and Access Delegation. Our approach (shown in Figure 2b) automates most of the process by: (**S1**) avoiding the need for reading online documentations: it provides a built-in list of IdMPs with their related information (e.g., endpoints); thus solving problem (P1) above; (**S2**) avoiding the need for downloading several SDKs: it is based only on the AppAuth SDK that is (effortlessly) integrated once and for all (solving P2); and (**S3**) automatically generating the code to integrate AppAuth within the app and enable the communication with multiple IdMPs (solving P3).

To summarize, our main contributions are three. First (to the best of our knowledge), we provide the first report about the compliance of popular IdMPs with the OAuth/OIDC best current practices for native apps. Our analysis reveals that, unfortunately, several IdMPs still do not follow the best practices. All the details of the report are available at `https://sites.google.com/fbk.eu/midassistant`. Second, we propose a novel wizard-based approach for secure code generation for mobile apps, allowing app developers to integrate multiple third-party IdMPs effortlessly and in a secure manner. In particular—by leveraging the information in a DB and a user-friendly guide for app developers—it provides a way to "enforce" the usage of best practices for the compliant IdMPs by using AppAuth, and securely interact with the IdMPs that are not fully-compliant. Third, we provide a prototype implementation of our approach as a plugin for Android Studio, and test it by interacting with several IdMPs, such as OKTA, Auth0, Microsoft, and Google.



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

Figure 2: The comparison between current and proposed approach.

Paper Structure. Section 2 introduces OAuth and OIDC, the de-facto standards for access delegation and SSO, respectively. Section 3 describes the current best practices for native apps. The proposed wizard-based approach for compliant and not fully-compliant IdMPs is explained in Section 4. We summarize the main results and provide some insights for future work in Section 5.

2 BACKGROUND ON OAUTH AND OPENID CONNECT

In the context of light-RESTful API services, OAuth and OIDC are the current de-facto standards for providing access delegation and SSO login, respectively.

2.1 Access Delegation: OAuth 2.0

OAuth (Hardt, 2012) is an authorization protocol that regulates what an entity is allowed to access. In simple words, it is a mechanism by which a user, called Resource Owner (*RO*), can delegate access for selected pieces of information hosted in a Resource Server (*RS*) to a designated Client (*C*) app, without having to share her credentials with C. Indeed, RO direct authenticates by using her User Agent (*UA*, e.g., a browser) 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 Work Group (WG) defines four grant types, which are dependent on the both involved entities and the relative scenario assumptions, which are: *RO Password Credentials*, *Implicit*, *Authorization Code*, and *Client Credentials*. The aforementioned grant types have a common goal that is releasing an *access token* to *C* for granting access to resources; differences lie in the process used to obtain tokens. Among these four grant types, *Implicit* and *Authorization code* flows are the most used by the apps. The *Implicit* grant is a flow optimized for *UA-based* apps, which are developed using a scripting language. However, the OAuth WG recommended not to use the Implicit grant flow as it is vulnerable to access token leakage and replay as detailed in the OAuth 2.0 Security Best Current Practice (Lodderstedt et al., 2019). The *Authorization Code* is the most popular grant type, which is suitable for both confidential (web) and public (native) apps. This flow possesses some security benefits in comparison with previous flows, such as the ability of *AS* to authenticate *C* and the direct transmission of the *access token* to *C* without passing it through *UA*, which is avoiding the exposure of the token to others, including *RO*.

2.2 SSO Login: OpenID Connect

OIDC (Sakimura et al., 2014) is an authentication layer developed on top of the OAuth standard. The problem that has led to introduce OIDC is the fact that OAuth protocol is often abused to implement authentication by major apps, while its main purpose is Access Delegation. In (Chen et al., 2014), the authors studied several vulnerabilities that raised by 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 (Jones et al., 2015) that contains information about the token issuer (the OpenID provider), the subject (user identifier) and the audience (the intended *C* app), all signed by the OpenID provider. This token enables *C* to safely verify that the received token is issued as result of its previous token request. While the *userInfo endpoint* is used to obtain identity-related attributes (e.g., the email and address) of the user.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

3 A CRITIQUE OF AVAILABLE BEST PRACTICES AND OUR FINDINGS

The goal of this section is to present the current best practices for SSO and Access Delegation for native apps and provide a brief description of our findings about IdMPs compatibility. In the following, firstly, the OAuth best practices for native apps are presented (cf. Section 3.1). Finally, a summary of our findings with regard to the IdMPs compatibility with the described best practice are displayed (cf. Section 3.2).

3.1 Best Practices for Native Apps

The lack of details in (Hardt, 2012) on how to implement OAuth for native apps has caused the spread of many insecure solutions. This is probably the reason why in 2017, the OAuth WG has released "OAuth 2.0 for Native Apps" (Denniss and Bradley, 2017), a set of best practices for implementing in a secure and usable way OAuth—and consequently OIDC given that is based on OAuth—in case the role of *C* is played by a native app. They suggest which (*i*) flow, (*ii*) *UA* and (*iii*) redirection mechanism must be used. More details are provided in the following sections and the key points are summarized in Table 1.

3.1.1 Flow: Authorization Code with PKCE

The suggested flow is the *Authorization Code* flow together with the Proof Key for Code Exchange (*PKCE*) protocol (Sakimura et al., 2015). The OAuth WG introduced *PKCE* to avoid the authorization code interception on public clients. If an attacker is able to steal an authorization code, then he can use it to obtain an access token and so access the user data. The *PKCE* avoids the aforementioned attack by adding three parameters: a code verifier (an unique secret of *C* app), a code challenge (e.g., hash of the code verifier) and the name of the transformation function used to calculate the challenge value. In simple words, the security introduced by the *PKCE* comes from the fact that no one rather than the legitimate *C* app should know the code verifier. This flow, illustrated in Figure 3, encompasses the following steps:

Table 1: Best practices for native apps.

Concept	Best Practice
Flow	Authorization Code w/o client secret
PKCE	If Custom URI, then PKCE mandatory
UA	External (possibly "in-app browser")
Redirection	HTTPS or Custom URI

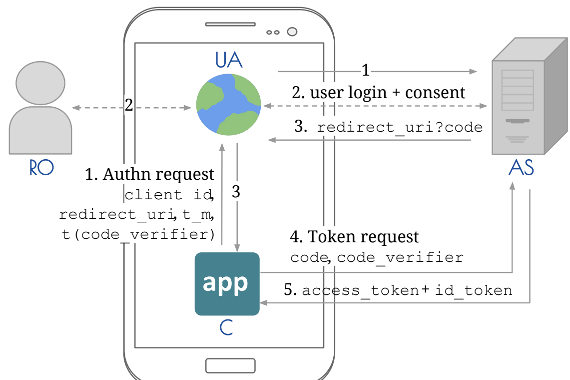


Figure 3: Authorization Code Flow with *PKCE*.

1. the *C* app initiates the flow by directing the *RO's UA* to the authorization endpoint. It sends, together with the OAuth parameters (*client_id*, *redirect_URI*), the *PKCE* parameters: $t(code_verifier)$ and *t_m*;
2. *AS* authenticates *RO* by displaying the login form (in *UA*) and establishes whether *RO* has granted or denied the access request;
3. assuming the access grant acceptance by *RO*, *AS* redirects *UA* back to *C* with an authorization *code*;
4. *C* sends the authorization *code* with *code_verifier* to *AS*;
5. in this step, *AS* computes the $t(code_verifier)$ value and compares it with the value received in Step 1. If they are the same, it releases an *access token* and *id token* to *C*.

In addition, as all information passing through a mobile device is visible to *RO*, the confidentiality of the *client_secret* is not guaranteed (a mobile 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.

3.1.2 User Agent: External Browser

In the Android environment, two kinds of *UA* are supported: *embedded* (e.g., Web View) and *external* (e.g., OS browsers or native apps). Based on the best practices, for security reasons, developers should avoid the usage of *embedded UA*, due to the full control of the app on the user data (a malicious *C* app could, for example, steal the *RO*'s password). Thus, the best practice recommends the usage of the *external UA*, in particular an external browser supporting the "in-App browser tabs" (Denniss and Bradley, 2017). This feature (known as Android Custom Tab in the Android platform) is a programmatic instantiation of the browser that keeps both security properties and authentication of the browser, while being displayed within the host app.

3.1.3 Redirection Method: Custom URI or HTTPS URI Scheme

For the redirection of the authorization *code* to the *C* app, there are three methods, which can be used within native apps. These methods can be categorized as *(i)* App-declared *Custom URI* scheme, *(ii)* App-claimed *HTTPS URI* scheme, and *(iii) Loopback URI* redirection (Denniss and Bradley, 2017). Among the aforementioned methods, App-claimed *HTTPS URI* redirection is recommended wherever it is supported by the operating system, because it provides a way to guarantee the identity of native apps, which is known as App Link. In this case, the OS checks, after the installation of the app, that the app has the right to claim a specific URL, in this way checking the app identity (Liu et al., 2017). Otherwise, *Custom URI* scheme is recommended—due to its wider adaptability with different Android OS versions—but it must be used alongside with *PKCE*.

3.1.4 AppAuth Library

AppAuth is an open source library for third-party apps to communicate with OAuth and OIDC providers, which is maintained by the OpenID foundation. It follows the best practices that are set out in (Denniss and Bradley, 2017). The current library can be downloaded, for both Android and iOS, at `https://appauth.io`.

3.2 IdMP Compatibility with AppAuth

We have selected a total of 15 IdMPs, from the lists that are available on the OIDC and OAuth websites,[1] based on their Alexa Rank, if they support the mobile native scenario and for which the developer console is accessible without subscription. We verified the compatibility of the selected IdMPs with AppAuth by integrating AppAuth SDK within a demo app. Our finding can be briefly summarized as follows:

- 5 IdMPs are not supported by the AppAuth due to the different grant types (e.g., Facebook and Spotify);
- 5 IdMPs are fully-compatible with the default setting of AppAuth (e.g., Google and OKTA). By using the AppAuth, the library enforces the best practices by avoiding using of implicit flow and embedded web view in Auth0 and Microsoft, respectively;

[1] https://www.openid.net/certification/ and https://oauth.net/code/. Last accessed Dec., 2018.

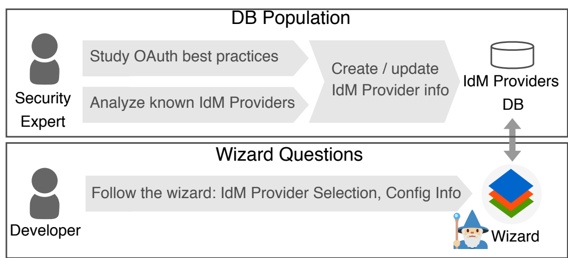


Figure 4: Approach.

- 5 IdMPs require some code changes like omitting *PKCE* and adding the client secret (e.g., Yahoo and Linkedin).

Even if the AppAuth SDK could be used with a specific IdMP, its integration in the use case scenario described in Section 1 is not easy and requires major changes: the developer should implement varied classes and methods to handle the communication with multiple providers. Furthermore, complementary details about the selection provedure, the metrics used and our findings with regard to the IdMPs compatibility are available at our companion website: https://sites.google.com/fbk.eu/midassistant.

4 WIZARD-BASED APPROACH

Our wizard-based approach encompasses the two processes depicted in Figure 4. In the former process, security experts or a community of them (e.g., members of the OAuth WG) populate the database of supported IdMPs. In the latter process, (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 compliant IdMP and not fully-compliant IdMP, respectively.

4.1 Support Compliant IdMP

We refer to *compliant IdMP* in case a provider follows the current best practices for SSO and access delegation for native apps (cf. Table 1). In our analysis: Google, Microsoft v2, OKTA and Auth0.

4.1.1 Security Experts: DB Population

Let us preliminary consider the information that security experts are required to provide for each new *IDMP*. At first, the security experts must evaluate whether *IDMP* is fully-compliant

Table 2: Example of IdM providers info.

	COMPLIANT		
Google	Scenario=both DevDomain=no	Redirect=both	Discovery=accounts.google.com/.well-known/openid-configuration
OKTA	Scenario=both DevDomain=yes	Redirect=both	Discovery=*DevDomain*/.well-known/openid-configuration
	NO COMPLIANT		
Yahoo	Scenario=both PKCE=no DevDomain=no	Redirect=https Secret=yes	Discovery=login.yahoo.com/.well-known/openid-configuration
Box	Scenario=AD PKCE=no DevDomain=no	Redirect=https Secret=yes	AuthnURL=https://account.box.com/api/oauth2/authorize TokenURL=https://account.box.com/api/oauth2/token

with AppAuth, namely if it follows what reported in Section 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: *IDMP* supports only HTTPS redirection (*IDMP*.Redirect=HTTPS), only Custom URI Scheme (*IDMP*.Redirect=CustomURI), or both (*IDMP*.Redirect=both). In addition, they should specify either the discovery URL (if supported) (*IDMP*.Discovery=*Value*), or their endpoints: Authorization Endpoint (*IDMP*.AuthnURL=*Value*) and Token Endpoint (*IDMP*.TokenURL=*Value*). 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 User Info Endpoint (*IDMP*.UserInfoURL=*Value*) must be provided as well. An example of DB population for two compliant *IDMPs* is provided in Table 2.

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.

4.1.2 Developers: Wizard Questions

In Listing1, we clarify which are the questions asked to an app developer through the wizard (in *italic*) and how they are customized according to the information about *IDMP*. 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. It is important to remark that the wizard helps the developer to select the proper flow based on the scenario to avoid for instance the wrong usage of OAuth, instead of OIDC, for authentication purposes.

4.2 Support not Fully-compliant IdMPs

The described wizard-based approach works for the IdMPs that are compliant with (Denniss and Bradley, 2017) and thus compatible with AppAuth SDK. To extend the capabilities of our wizard in order to support other IdMPs that can be still used in a secure manner (referred as *not fully-compliant IdMPs*), by leveraging the results of the analysis, we identified the following changes: *(i)* omit the sending of the *PKCE* parameters whenever the IdMP's server does not support the *PKCE* protocol, but at the same time enforce the *HTTPS Redirection* scheme within the wizard, and *(ii)* enforce the sending of the Client Secret whenever its value is not actually used as a secret. The best practices for the native apps recommend to avoid using 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 has to verify that the Client Secret used for a mobile native app is, at least, different from the one released for web apps. In the case that the IdMPs rely on the "secrecy" of the Client Secret, as future work we plan to: require the dynamic client registration, in such a way to have a different Client Secret for app installation; or perform the code exchange step though the backend of the mobile app (if it is present) that can store the Client Secret.

4.2.1 Security Experts: DB Population

As a consequence, as shown in Table 2, in case an IdMP is not fully-compliant with AppAuth, the security experts must evaluate whether it is due to:

Listing 1: Wizard for compliant IdMPs.

```
1  // Scenario
2  1. Choose whether you want to add in your app a
       button for either [1.1] SSO Login, or [1.2] AD
3  if [1.1] then
4  the wizard shows a list of all IDMP such that
       IDMP.Scenario=both or IDMP.Scenario=sso
5  if [1.2] then
6  the wizard shows a list of all IDMP such that
       IDMP.Scenario=both or IDMP.Scenario=AD
7
8  // Selection
9  2. Please, select IDMP among the list of supported
       IdM Providers
10 The wizard reads from the IdM Providers DB the
       IDMP endpoints and fills the AppAuth conf file.
11
12 // Configuration
13 3. Enter Configuration Info: Scopes (optional),
       ClientID, the Name of the Button you want to add
14 The provided info is used by the wizard to update the
       AppAuth conf file and create the button.
15
16 If [1.2] then
17 Please, enter the resource endpoint
18 The provided endpoint is used by the wizard to update
       the AppAuth conf file.
19
20 If IDMP.DevDomain=yes then
21 Please, enter your developer domain
22 The provided endpoint is used by the wizard to update
       the AppAuth conf file.
23
24 // Redirection
25 If IDMP.Redirect=both then
26 4. Choose the preferred Redirection Method: either
       [4.1] HTTPS scheme, or [4.2] CustomURI
       scheme
27 If [4.1] or IDMP.Redirect=HTTPS then
28 Enter your valid domain URL: scheme, host, and path
29 If [4.2] or IDMP.Redirect=CustomURI then
30 Enter the Custom URI of your app
31 The provided info is used by the wizard to complete
       the AppAuth conf file and fill the Intent filter in
       the Android manifest.
```

- the lack of *PKCE* and the IdMP supports HTTPS Redirect, and if this is the case it 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 it should specify *IDMP*.`Secret=yes`.

4.2.2 Developers: Wizard Questions

Concerning the changes in the wizard, while the Scenario, Selection and Configuration parts are the same of Listing 1, in Listing 2, we clarify how the other questions change in case *IDMP* is not fully-compliant. In detail: in lines 1-4, if *IDMP* does not support the *PKCE* protocol then we ask for an HTTPS

Listing 2: Wizard for not fully-compliant IdMPs.

```
1  // Redirection
2  If IDMP.PKCE=no then
3  Enter your valid domain URL: scheme, host, and path
4  Otherwise same behaviour of Listing 1, lines 20-27
5
6  // Client Secret
7  If IDMP.Secret=yes then
8  Enter the App Client Secret
```

URI scheme, while in lines 6-8, if *IDMP* requires a Client Secret, its value is asked to the developer.

4.3 Prototype and Experimental Results

To assess the effectiveness of our approach, we developed mIDAssistant, a prototype implementing our approach for Android OS to support native apps. mIDAssistant is an open-source Android Studio plugin developed using the Intellij Idea environment, and it can be integrated in the development environment just with few clicks. The plugin leverages the AppAuth library as a main core to guarantee the enforcement of the best practices. On top of that, different methodologies—such as File Template and Program Structure Interface (PSI)—are used in order to enable the plugin to generate and automatically insert the code within the proper location of developer's app. We have used mIDAssistant to implement our use-case scenario within a demo app, which is capable of communicating with Auth0, OKTA, Google and Microsoft as they are compatible with AppAuth, and with Yahoo, Linkedin, DropBox, Buffer, and Box after the changes explained in Section 4.2.

5 DISCUSSION AND FUTURE WORK

To the best of our knowledge, the state-of-the-art methods are not strictly in the same line as the work presented. There are some works, such as Chex (Lu et al., 2012), RoleCast (Son et al., 2011), Android Lint Tool (Google, 2016), FixDroid (Nguyen et al., 2017), and App Link Assistant (Google, 2017) that help app developers to implement secure code, but none of the aforementioned works help developers with multiple IdMPs integration. Concerning other tools supporting AppAuth, it is worth mentioning that Xamarin (Xamarin, 2015)—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.

To conclude, in this paper we have discussed how it is time-consuming and error-prone for inexperienced app developers to integrate several IdMPs (e.g., using Google for SSO login and Microsoft OneDrive for managing the document Access Delegation). To this end, we propose a novel wizard-based approach that guides developers through integration of multiple third-party IdMPs within their app, by *(i)* "enforcing" the usage of the best practices for native apps (AppAuth support), *(ii)* avoiding the need for download several SDKs and reading their online documentations (a list of known IdMPs with their configuration are embedded within our approach), and *(iii)* automatically generating the required code. A prototype of our approach has been implemented as an Android Studio plugin. It is currently capable of speaking with Google, OKTA, Auth0, and Microsoft, which enforce the best practices, and with not fully-compliant IdMPs such as Yahoo, DropBox, Box, Linkedin, and Buffer. It is worth mentioning that, during the OAuth Security Workshop 2019, we spoke with members of the OpenID Foundation about possible collaborations and they have shown interest on our approach.

As a future work, we plan to: *(i)* expand our analysis by considering other popular IdMPs; *(ii)* evaluate the possibility to provide an automatic way to extract the information for the new IdMP; *(iii)* design and conduct a user-study experiment to evaluate our approach; and *(iv)* add the code exchange on the app backend to secure cases that need to use the client secret during the authorization request.

REFERENCES

Chen, E. Y., Pei, Y., Chen, S., Tian, Y., Kotcher, R., and Tague, P. (2014). OAuth Demystified for Mobile Application Developers. In *ACM CCS*.

Denniss, W. and Bradley, J. (2017). OAuth 2.0 for Native Apps. IETF.

Google (2016). Android Lint. https://developer.android.com/studio/write/lint.

Google (2017). App Link Assistant Tool. https://developer.android.com/studio/write/app-link-indexing.

Hardt, D. (2012). The OAuth 2.0 Authorization Framework. IETF.

Jones, M., Bradley, J., and Sakimura, N. (2015). JSON Web Token (JWT). IETF.

Liu, F., Wang, C., Pico, A., Yao, D., and Wang, G. (2017). Measuring the Insecurity of Mobile Deep Links of Android. In *USENIX Security'17*.

Liu, X., Liu, J., Wang, W., and Zhu, S. (2018). Android Single Sign-On Security: Issues, Taxonomy and Directions. *Future Generation Computer Systems*, 89:402–420.

Lodderstedt, T., Bradley, J., Labunets, A., and Fett, D. (2019). OAuth 2.0 Security Best Current Practice.

Lu, L., Li, Z., Wu, Z., Lee, W., and Jiang, G. (2012). Chex: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *ACM CCS*, pages 229–240.

Nguyen, D. C., Wermke, D., Acar, Y., Backes, M., Weir, C., and Fahl, S. (2017). A stitch in time: Supporting Android developers in writing secure code. In *ACM CCS*, pages 1065–1077.

Sakimura, N., Bradley, J., and Agarwal, N. (2015). Proof Key for Code Exchange by OAuth Public Clients. IETF.

Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and Mortimore, C. (2014). OpenID Connect Core 1.0 incorporating errata set 1. OIDF.

Son, S., McKinley, K. S., and Shmatikov, V. (2011). Rolecast: finding missing security checks when you do not know what checks are. In *ACM Sigplan Notices*, volume 46. ACM.

Xamarin (2015). Xamarin Tools for cross platform app development. https://releases.xamarin.com.

Yang, R., Lau, W. C., and Shi, S. (2017). Breaking and Fixing Mobile App Authentication with OAuth2.0-based Protocols. In *ACNS*, pages 313–335. Springer.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS