Security of Mobile Single Sign-On: A Rational Reconstruction of Facebook Login Solution

Giada Sciarretta[1,2], Alessandro Armando[1,3], Roberto Carbone[1] and Silvio Ranise[1]
[1]*Security & Trust, FBK-Irst, Trento, Italia*
[2]*University of Trento, Trento, Italia*
[3]*DIBRIS, University of Genova, Genova, Italia*

Keywords: Single Sign-On, Digital Identity, Security of Mobile Devices, OAuth 2.0.

Abstract: While there exist many secure authentication and authorization solutions for web applications, their adaptation in the mobile context is a new and open challenge. In this paper, we argue that the lack of a proper reference model for Single Sign-On (SSO) for mobile native applications drives many social network vendors (acting as Identity Providers) to develop their own mobile solution. However, as the implementation details are not well documented, it is difficult to establish the proper security level of these solutions. We thus provide a rational reconstruction of the Facebook SSO flow, including a comparison with the OAuth 2.0 standard and a security analysis obtained testing the Facebook SSO reconstruction against a set of identified SSO attacks. Based on this analysis, we have modified and generalized the Facebook solution proposing a native SSO solution capable of solving the identified vulnerabilities and accommodating any Identity Provider.

1 INTRODUCTION

Single Sign-On (SSO) protocols are arguably one of the most successful security solutions available today. They allow users to access multiple services through a single authentication act carried out with an authentication server acting as an Identity Provider (IdP). Note that, by reducing the number of digital identities (and credentials) a user has to deal with, SSO protocols improve the user's experience and security (e.g., stronger password selection).

While for web applications there exist many secure authentication and authorization solutions to protect the user's digital identities and online resources, solutions for mobile applications are not yet consolidated. The existing protocols, such as OAuth 2.0 (IETF, 2012a) and OpenID Connect (OIDF, 2014a), provide only a partial support for mobile native applications, leaving many implementation choices to developers. Leveraging these protocols, many social networks have already deployed their own authentication and authorization solutions, which have been tremendously successful: most Facebook and Google users routinely and transparently use them on their smartphones and tablets. However, the adaptation of protocols originally designed to work in a traditional web scenario, together with the lack of a complete SSO standard for mobile native applications (hereafter native SSO)—it is only available a draft version released by the working group NAPPS (OIDF, 2014b) of the OpenID Foundation—have caused the spread of a number of serious vulnerabilities and attacks. There are many studies in the literature, such as (Chen et al., 2014; Wang et al., 2013; Shehab and Mohsen, 2014), which focus on the analysis and description of common vulnerabilities and attacks caused by incorrect implementation assumptions; however—to the best of our knowledge—it remains unclear how to implement native SSO solutions in a secure way.

This paper provides a model which can be used to support the native SSO development. More specifically, we make the following three contributions. First, we provide a rational reconstruction of the Facebook[1] native SSO. Second, we include detailed security considerations for the Facebook solution. Third, we propose a native SSO solution inspired by the Facebook native SSO but capable of solving the identified vulnerabilities.

[1]In the rest of this paper we will focus on the SSO solution used by Facebook. However the concepts discussed in this paper can be easily adapted to the SSO solution developed by Google.

Sciarretta, G., Armando, A., Carbone, R. and Ranise, S.
Security of Mobile Single Sign-On: A Rational Reconstruction of Facebook Login Solution.
DOI: 10.5220/0005969001470158
In *Proceedings of the 13th International Joint Conference on e-Business and Telecommunications (ICETE 2016) - Volume 4: SECRYPT*, pages 147-158
ISBN: 978-989-758-196-0

The rest of the paper is structured as follows. In Section 2, we describe the basic notions of a SSO scenario and, in the context of native applications, we give a short overview of the OAuth 2.0 protocol. We also provide an application scenario for the proposed solution. In Section 3, we detail our rational reconstruction and security analysis of the Facebook solution. Together with the description of the flow, we make a comparison with OAuth flows and explain Facebook native SSO security issues. In Section 4, we describe our solution for native SSO, comparing it with the Facebook solution. Finally, we present the related work in Section 5, and we draw our conclusions and discuss our future work in Section 6. In addition, in Appendix we detail the flow of our native SSO solution.

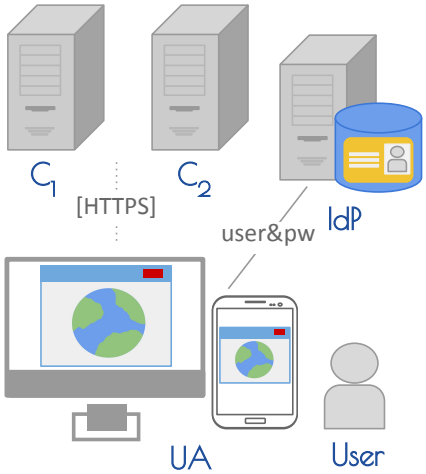


Figure 1: SSO for Web Apps.

2 BACKGROUND

The goal of this section is to provide the basic notions required to clearly understand the analysis performed on the Facebook native SSO solution. Section 2.1 describes the entities involved and the functional requirements of a SSO process. In Section 2.2, we give an overview of the OAuth 2.0 protocol, used by Facebook to perform the login process.

2.1 SSO: Entities and Requirements

SSO protocols allow users to access multiple services through a single authentication act. The *User* participates in the protocol through a *User Agent* (UA) and authenticates with an *Identity Provider* (IdP) with the purpose of proving her identity to a *Client* entity (C). We assume a SSO solution meets the following two requirements:

(R1) the IdP user credentials can be used to gain access to several Cs. This implies that users do not need to have credentials with a C to access it.

(R2) If a user has already a login session with an IdP, then she can access new Cs without entering her IdP credentials anymore, and only the user consent is required.

A typical SSO scenario consists of the following steps: first a user asks for access to C. Then C issues an authentication requests to IdP (through UA). The IdP authenticates the user and establishes a security context with her. The IdP then sends an authentication assertion containing the user information to C passing back through UA. Whenever in the future the user will access new Cs through the same UA supporting the login with the same IdP the login phase will be skipped, thereby satisfying (R2).

The SSO entities are instantiated in different ways based on the scenario and technology used. As we will describe in Section 5, while the SSO scenario where C apps are web application (illustrated in Figure 1) is very studied, the case where Cs are native applications is quite new and challenging. For this reason, in the rest of this paper we will focus on the authentication process for Android[2] native apps.

2.2 OAuth for Native Applications

Many standards and products on authentication and authorization have been developed in the last few years by prominent organizations. Here we present the OAuth 2.0 (OAuth, for short) (IETF, 2012a), which is one of the most widespread protocols used in the case of mobile native apps.

OAuth is an authorization protocol typically used to manage delegation of authorization. To illustrate consider a user, called Resource Owner (RO), which wants to use an app, called Client (C), to print some photos (resources) stored on a different photo-sharing site (resource server). Using OAuth, RO can grant C access to her private photos, without having to share her credentials by directly authenticating with a server trusted by the photo-sharing service, called Authorization Server (AS), which issues an access token to C carrying the requested authorization delegation. The default access token type used in the OAuth protocol is a *bearer token*, which in (IETF, 2012b) is defined as "a security token with the property that any party in possession of the token (a "bearer") can use

[2]We focus on Android, even if many concepts can be applied on other operating systems.

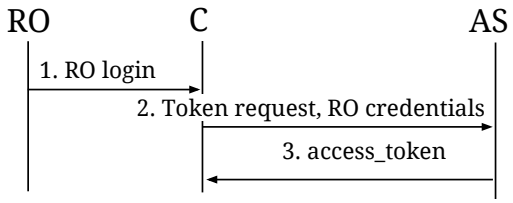


Figure 2: OAuth RO Password Credential Flow.

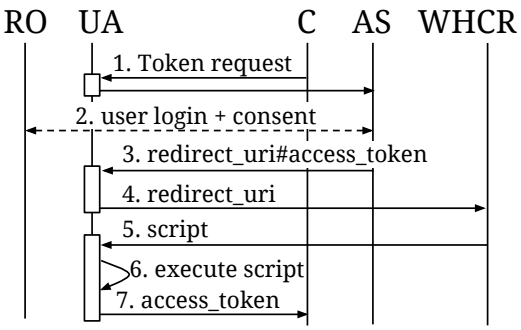


Figure 3: OAuth Implicit Flow.

the token in any way that any other party in possession of it can. Using a bearer token does not require a bearer to prove possession of cryptographic key material (proof-of-possession)". This means that an entity in possession of an access token can access the corresponding RO resources without any further authentication and authorization process.

In many implementations, OAuth is also used for authentication by assuming that the resource that C wants to access is the user profile on AS. If C can access a user profile, this means that the RO has granted the permission. This authorization act is used by C as a proof of the user identity.

In the case of native apps OAuth does not specify which flow to use. We have three possibilities: *Authorization Code*, *Implicit* and *Resource Owner Password Credentials*. Note that, the Authorization Code is the only flow that requires the authentication of C. Indeed, in the final step of the flow C exchanges a *code* and its *client secret* to obtain an access token in a direct communication with AS. The client secret is a value obtained in the registration phase and known only by itself. In the case of native apps, as every information is visible by the RO (the owner of the smartphone where the native apps are installed), the confidentiality of the client secret is not guaranteed. These types of C, for which the client secret is not confidential, are called *public* Cs. Note that, if every information pass through the RO's smartphone, the exchange of the client secret for an access token just makes the flow more complex without adding more security. Thus, we will detail only the flows designed to be used with public Cs.

RO Password Credentials Flow. This flow, illustrated in Figure 2, includes the following steps: in Step 1, RO provides C with her own credentials. Then, in Step 2, C requests an access token from AS's token endpoint by including the credentials received from RO. Finally, in Step 3, AS validates the RO credentials, and if valid, issues an access token.

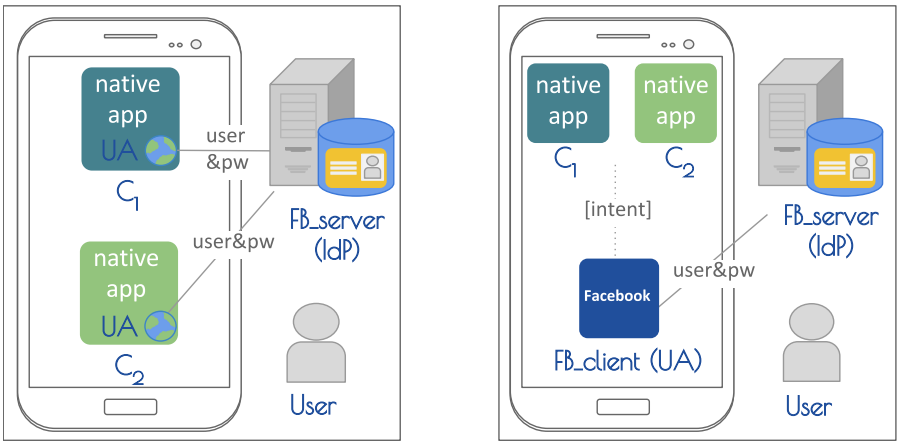
Implicit Grant Flow. This flow, illustrated in Figure 3, involves the following steps: in Step 1, C initiates the flow by directing RO's UA to the authorization endpoint. C also includes a redirection URI to which AS will send UA back once access is granted (or denied). In Step 2, AS authenticates RO (via UA) and establishes whether RO grants or denies C's access request. Assuming RO grants access, AS redirects UA back to C (Step 3). The redirection URI includes the access token in the URI fragment. In Step 4, UA follows the redirection instructions by making a request (which does not include the fragment) to Web-Hosted Client Resource (WHCR). UA retains the fragment information locally. In Step 5, WHCR returns a web page (typically an HTML document with an embedded script) capable of accessing the full redirection URI including the fragment retained by UA. UA executes the script locally (Step 6), which extracts the access token. In Step 7, UA passes the access token to C.

2.3 Application Scenario

In this section, to motivate our work and give an illustrative example of the applicability of the proposed native SSO solution, we describe an application scenario that involves e-health mobile applications.

TreC (acronym for Cartella Clinica del Cittadino, i.e. Citizens' Clinical Record) is a platform[3] developed in the Trentino region for managing personal health records. Besides the web platform, which is routinely used by around 61,168 users since 2012, TreC is currently designing and implementing a number of native Android applications to support self-management and remote monitoring of chronic conditions. These applications are used in "living lab" context by voluntary chronic patients according to their hospital physicians. An example is the "TreC-Lab: Diario Diabete" app, which is a mobile diary that allows patients to record health data, such as the blood glucose level and physical activity. While in the traditional web scenario, patients access services using

[3]The development of the platform is supported by a joint project between Fondazione Bruno Kessler and Municipality of Trento (Italy). More information is available at https://trec.trentinosalute.net/.



(a) Native Apps and Embedded Browsers. (b) Native Apps Only.

Figure 4: Facebook Native SSO Solutions.

their provincial health care system credentials (leveraging a SAML-based SSO (OASIS, 2005) solution), a solution for native SSO is currently missing.

The solution we propose will allow patients to access different TreC e-health mobile applications (and possibly other third-party e-health applications) through a single authentication act with the provincial health care system. In particular, the different entities of SSO solution will be instantiated in the following way: C apps will be the TreC e-health mobile applications, the User will be a patient that wants to use the TreC apps to manage her health data, and the IdP will be played by the IdP of the provincial health care system. In addition, being the TreC data sensitive ones, the proposed solution must be also flexible enough to support strong authentication mechanisms.

As we will show in the next sections, our solution is an excellent starting point to fulfill the TreC needs and accommodate the provincial health care system.

3 FACEBOOK NATIVE SSO: A RATIONAL RECONSTRUCTION

Facebook (FB, for short) exploits OAuth for both authentication and authorization. In this paper, we will focus on the authentication process for Android native apps, which allows app developers to integrate the FB login feature, and so exploit the FB native SSO solution. In Section 3.1, we describe the different FB native SSO solutions identifying the scenario that we will take into account during our security analysis. Section 3.2 describes our rational reconstruction of FB native SSO. Together with the details of the flow, we make a comparison with OAuth flows. In Section 3.3, we define the assumptions and the threat model that we will use in the FB security analysis, and finally, in Section 3.4, our security considerations on FB solution are presented. The result of our analysis shows that the FB native SSO is vulnerable to known SSO attacks.

3.1 Facebook Native SSO Solutions

FB provides a Software Development Kit (SDK) to help mobile app developers to implement the login feature. As reported in the FB login guide (Facebook, 2015), the SDK has two different kinds of login implementation where the SSO entities are instantiated in different ways: *(i)* in Figure 4(a), C_1 and C_2[4] are native apps installed on user smartphones, UAs are embedded in the apps, and the IdP is played by FB_server; *(ii)* in Figure 4(b), C_1 and C_2 are native apps installed on user smartphones, UA is a FB Android app (hereafter FB_client) handling user authentication locally on the device, and the IdP is played by FB_server.

As pointed out in (Shehab and Mohsen, 2014) and (Boyd, 2012), if an embedded browser is used (case of Figure 4(a)), C has full control of the hosted UA. Thus, if C is malicious, then it can steal the user credentials or change the authorization permissions. An additional limitation for the embedded browser solution is that it does not satisfy the requirement (R2). Indeed, if the browser is integrated within the app, then the login session information is saved in (and only accessible to) the app and it is therefore not available to other apps. This forces the user to re-enter credentials even if she has active login sessions with

[4]For the sake of simplicity we consider only two Cs, the generalization to more than two Cs is straightforward.

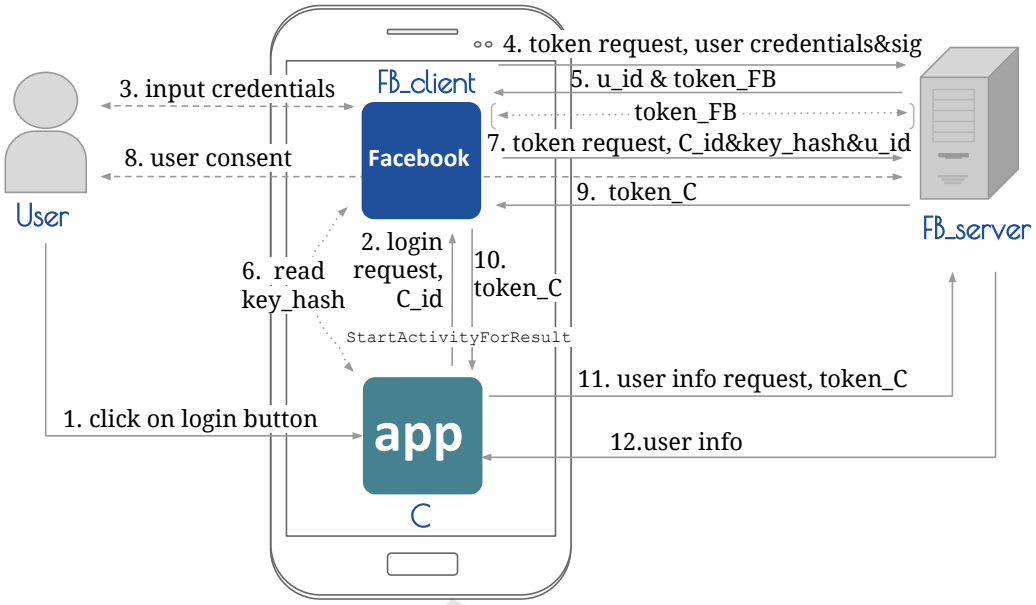


Figure 5: Facebook Native SSO Solution.

an IdP. We thus disregard this solution and take into account the solution shown in Figure 4(b).

3.2 Facebook Flow Description

To reconstruct the FB native SSO solution, we implemented an application that integrates the FB Login feature following the FB documentation (Facebook, 2015) and using the FB Android SDK. At the time of our analysis, we used Android 4.3.1, FB Android SDK 3.22.0 and FB APK 23.0.0.22.14.[5]

To obtain a precise description of the FB native SSO flow, we have performed the following methodology: *(i)* we have analyzed the source code of the FB SDK to understand the interaction between our app and FB_client inside the smartphone, and *(ii)* we have used the Fiddler proxy tool[6] to carefully inspect the HTTP(S) traffic between the FB_client and the FB_server.

Registration Phase. To use the FB login solution we had to register with FB. In this phase, we have entered the app package name and the certificate fingerprint (called, *key_hash*) of our app (C). The key_hash is a digest (SHA1) of the file CERT.RSA, that contains the public key of the developer, the signature of the app package (APK) obtained with the private key of the developer, and other information about the certificate. After the registration phase, an identifier (*C_id*) is associated with our native app.

Before the step description, we define the Android components used by the FB solution. To manage the inter-communication between two apps, FB use explicit intents and the `startActivityForResult`[7] method. An *intent* is an inter-process communication mechanism and is called: *(i) explicit*, when the sender specifies the name of the receiver, and *(ii) implicit*, whereby the sender only specifies the required task without specifying any receiver. The `startActivityForResult` method allows an app to display an Activity (basically a user interface) of another app and get a result back.

SSO Phase. Figure 5 depicts the most important steps, abstracting away the steps that are irrelevant for the authentication process. In Step 1 the user opens C and clicks the "Login with Facebook" button to authenticate in C. This triggers the invocation (Step 2) of an Activity from FB_client using the method `startActivityForResult`. C is put on wait for the Activity to return a result. In Step 3, FB_client presents to the user an interface for prompting her credentials. The user enters her credentials and activates a "Log in" button. In Step 4, FB_client interacts with the FB_server in order to authenticate the user. Note

[5]We also analyzed the FB flow with more recent versions (Android 5.1.1, FB Android SDK 4.1.0 and FB APK 44.0.0.26.142), even if some parameters of the API calls are different, the overall flow matches.

[6]Available at http://fiddler2.com/.

[7]For more details see the Android documentation, available at http://developer.android.com/reference/android/app/Activity.html#startActivityForResult(android.content.Intent,int).

that, the HTTPS request contains the user credentials and a parameter, *sig*, which is the hash of the parameters on the request calculated using a specific value stored in the FB_client app. As a response to the authentication request (Step 5), FB_client gets the user information, among which there is the user's identifier (*u_id*), and an access_token (*token_FB*). token_FB is used by the FB_client to obtain configuration and user information. Note that, token_FB has no expiration date, but can be invalidated by the user either by changing password or logging-out in the device.

Remark 1. We can observe that Steps 3-5 of Figure 5 correspond to the steps of Figure 2 with the User, FB_client and FB_server playing the role of OAuth RO, C and AS respectively. As the RO password credential, the FB implementation eliminates the need for storing the user credentials for future use, by exchanging the credentials with a long-lived access token (token_FB).

In Step 6, using the Android method `getPackageInfo(client_packageName, PackageManager.GET_SIGNATURES)` FB_client extracts the information about the certificate fingerprint (key_hash) included in the package of C, and in Step 7, it sends a token request for C to FB_server, including C_id, key_hash and u_id. First, FB_server checks whether the provided key_hash matches the one previously provided by the developer for the given C_id in the registration phase. If there is no match then the flow is interrupted with an error, otherwise FB_server checks whether the user authorized C. If C was not authorized by the user, the response (Step 8) contains a consent form. The consent form asks the user to grant or deny C to read the profile information. Otherwise, if C was already authorized by the user, the consent phase is skipped. If the user agrees, the response (Step 9) contains the access_token for C (token_C). In Step 10, FB_client returns control to C and provides token_C as result of the invoked Activity (see Step 2).

Remark 2. At this point, it is interesting to do another comparison with the OAuth flows. We have that Steps 2 and 7-10 of Figure 5 can be compared with Steps 1-3 and 6-7 of Figure 3 with the User, C, FB_client, and FB_server playing the role of OAuth RO, C, UA, and AS, respectively. We can observe that in the FB solution the WHCR entity is missing. The WHCR is the server side of C and is needed to give to the UA the instructions to interpret the fragment values (e.g., access token value) returned in Step 3 and pass them to C. This procedure has been designed to manage the *same-origin policy* when Cs are web browser applications. Many native apps do not have a corresponding server, so in the FB native solution, the redirection URI is the same for every native app and has the value `fbconnect://success`. Indeed, it is the FB_client that received the script and read the access token out of the URI. In addition, we observe that Step 8 of Figure 5 corresponds to Step 2 of Figure 3 without the user login (the user identity is passed using the u_id parameter).

Finally, C sends a request (Step 11) to the FB_server including token_C to obtain the user information. C obtains the user information from FB and this proves that C is under control of the corresponding user.

By analyzing the FB solution, we have noticed that it is the combination of two OAuth flows (Resource Owner password credential and Implicit), and the FB_client app has two purposes: *(i)* it permits FB to authenticate C, and *(ii)* manages the SSO process (user authentication and access token release). In addition, we observe that the FB solution satisfies the requirements expected for a SSO solution: *(i)* a user has to register only with the FB_server and can use her FB credentials with different native apps (Cs). This satisfies requirements (R1); *(ii)* using FB_client allows user to enter her FB credentials just once, and the app will store the authenticated session. Every time a native app requires to login with the FB_server, the FB_client will employ the stored session without asking the user for credentials again. This satisfies (R2).

3.3 Security Assumptions and Threat Model

In order to perform a security analysis of the Facebook solution, we need to identify security assumptions and threat model of a native SSO flow. We decide to express these assumptions for a generalized SSO solution, where the entities involved are: the human participant (*User*), the server that authenticates the user (*IdP_server*), the native app that manages authentication and token exchange (*IdP_client*), and the native app that uses the authentication assertions of the IdP (*C*).

3.3.1 Assumptions

The exchange of authentication assertions among the different entities are regulated by the following assumptions:

(A1) IdP_server is trusted by C on identity assertions;

(A2) users download and install the proper IdP_client apps (i.e., IdP_client is authentic and

trusted);

(A3) messages between C/IdP_client and IdP_server are exchanged over HTTPS channels.

If (A1) does not hold, then IdP_server could be malicious and C may obtain a fake identity assertion and login the wrong user. If (A2) is not satisfied, the user credentials can be stolen by malicious IdP_client apps. Finally, (A3) is needed to avoid that a malicious attacker can sniff or change information during the network communication.

Besides the usual HTTP(S) channels used to interact with the remote entities, it is necessary to consider the communication mechanisms used inside the mobile device. In FB native SSO solution, as we have described in Section 3.2, the communication between two apps is performed using explicit intents and the `startActivityForResult` method. Note that, assuming

(A4) the integrity of the Android OS,

the Android method, used in FB, cannot be attacked. Indeed, A4 ensures: *(i)* app isolation, i.e. the data of an app cannot be visible to other apps on the device, *(ii)* protection against database corruption, i.e. an app cannot modify files of another app, and *(iii)* secure communication mechanisms, i.e., for example, an app cannot intercept or modify the value released as result of an Activity invoked by another app using the `startActivityForResult` method. In addition, as FB, we assume:

(A5) the use of explicit intents.

In this way, the system is not vulnerable to a number of known attacks described in (Chin et al., 2011).

Note that, assumptions (A4) and (A5) ensure: *(i)* the confidentiality of the messages sent from C to IdP_client, and *(ii)* the authenticity of the messages received by C from IdP_client.

3.3.2 Threat Model

Based on the assumptions defined above, an attacker may be:

Malicious App: an application installed on the users device that tries to violate the SSO flow. In particular, if the Malicious App plays the role of C in the SSO flow, we call it *Malicious C*. Compare to a general Malicious App, a Malicious C can also access security relevant parameters used in the SSO flow.

Malicious User: a human person that plays the role of User in the SSO flow and can extract information from her smartphone (e.g., having root permissions) or use tools to reverse engineering applications.

Clearly, an attacker can be also a combination of these attacker types.

3.4 Security Analysis of Facebook Native SSO

Given that the FB native SSO solution is based on OAuth, to perform our security analysis we took into account the security considerations reported in the OAuth specification (IETF, 2012a, §10), where for each possible attack the respective countermeasures are described. Our goal was to check whether the FB native SSO solution described in Section 3.2—under the security assumptions and thread models considered in Section 3.3—is vulnerable to the attacks reported in (IETF, 2012a). To test whether the attacks could be reproduced in the actual deployment, we used the implementation described in (Facebook, 2015), which requires the use of FB Android SDK, and playing the role of a malicious app. As a result of these tests, we have discovered that, even if some implementation choices (e.g., the use of explicit intent described in Section 3.3.1) protect the FB native SSO solution from some known attacks, it anyway suffers from:

- *Phishing*: malicious apps can create a fake login form and steal user credentials.

  This attack can be easily performed (as pointed out in (Khorana, nd)). In this case App_T, when the user clicks on the "Login with Facebook" button, shows a fake login form, equals to the one shown by the FB_client, and it is thus able to steal the victim's credentials. This is possible as the FB solution allows us to start the authentication process from the C apps. Note that, even assuming that the user is a security expert, she has no way to discover the ongoing attack as the login form shown by App_T can be identical to the login form shown in the FB login flow.

- *User Impersonation*: as a consequence of this attack, an attacker can login in a benign C as another user.

  To reproduce this vulnerability, we have performed the attack described in Figure 6, where we play the role of an attacker and Alice is the victim:

  – *Session 1:* App_T (playing the role of a C), which is installed in the Alice's smartphone, obtained a valid token linked to Alice's identity.

  – *Session 2:* we (playing the role of a user) obtained a new token for a benign C using our smartphone. Then, before the execution of Step 11, we changed the token linked to our identity

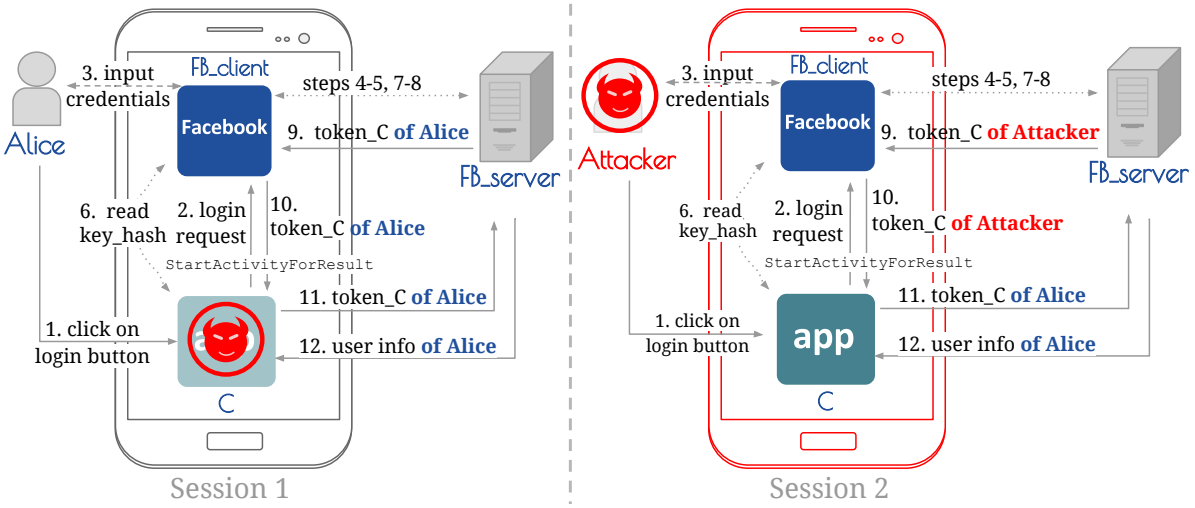


Figure 6: User Impersonation Attack in the Facebook Native SSO Solution.

with the Alice's token. This is possible as we (the attacker) are the owner of the smartphone where the SSO flow is running.

To mitigate the user impersonation, a C developer has to check that the token (or in general the identity assertion) used to retrieve the user information has been granted to his/her application. In the FB native SSO solution, the default token used is a bearer token, thus it does not contain any information about the authentication process, the only way to validate the token is to request the token information to the FB_server. The problem is that, as before, the attacker can manipulate this request. For certain types of apps (e.g., games), FB suggests to set as *response_token* parameter in Step 7 of FB native SSO flow the value *signed_request* (Facebook, 2016) (we do not write this parameter in Figure 5 as it is not related to the security analysis). Using this value, a C app can obtain extra information about users, such as the age and the origin country. In this way, before the real use, the C can personalize the app or deny the access (e.g., for game that requires a minimum age). This extra info is returned in a *Json Web Token* (JWT - (IETF, 2015)). We want to observe that, even using this token, the attack is valid. This is because the information about the audience of the token (the client identifier) is missing. So, a developer can check that a token was sent by FB and extract the user info, but cannot check if the token was released for itself.

- *Client Impersonation*: a malicious C can impersonate another C to the IdP_server obtaining user identities and access protected resources.

  As previously explained in Section 3.2, the FB solution is a combination of two OAuth flows, thus we have to consider two different clients:

  – the mobile app (C) in the Implicit flow, and
  – the FB_client plays the role of C in the RO password credential flow.

  In the first case, we have observed that, performing the verification of the key_hash value (after Step 7 in Figure 5), the FB solution mitigates the client impersonation attack. Indeed, a malicious C cannot have the same certificate of a benign C, as the certificate depends on the private key of the C developer.

  In the second case (when FB_client plays the role of C), we discovered that the client impersonation attack is exploitable. This is possible as the knowledge of the user's credentials is the only factor used by the FB_server to authenticate the FB_client. Thus, being the FB solution vulnerable to phishing attack, a malicious C, after obtaining the user's credentials, can also impersonate the FB_client sending the same message of Step 4 in Figure 5. To test this, we used the Advanced Rest Client,[8] allowing us to test custom HTTP requests. In detail, we have simulated the call of Step 4 of Figure 5 and, entering the user credentials, we were able to obtain a valid token. Note that, to perform this call, an attacker has to generate the *sig* parameter. In our test, we was able to discover the method used to generate the hash value by reverse-engineering the FB_client app. It is also important to notice that the token obtained was targeted for the FB_client, and so it has no

[8] Available at https://chrome.google.com/webstore/detail/advanced-rest-client/hgmloofddffdnphfgcellkdfbfbjeloo.

expiration date and it allows one to access all the user data.

4 SECURING THE FACEBOOK SSO SOLUTION

Starting from the FB native SSO solution and taking advantage of our security analysis, in this section we propose a native SSO solution capable of mitigating the vulnerabilities described in Section 3.4. This solution generalizes the one proposed by FB, in such a way that it can be used as a reference model by any IdP, willing to provide its own SSO solution.

Our generalized SSO solution involves the entities described in Section 3.3. The idea is that IdP_client enables the execution of the processes of (delegated) authorization and (user) authentication, needed by C to access the user profile, by exchanging messages with the IdP_server. After downloading IdP_client, both a *registration phase* and an *activation phase* are required before the actual execution of the native SSO flow. Below, we first hint the two preliminary phases and then describe the routine use of IdP_client, called *SSO process*.

Registration Phase. The goal is to enable C to interact with IdP_client to obtain the user profile. This registration phase is performed by the C developers and being in line with the FB registration phase we have detailed it in Appendix.

Activation Phase. The goal is to enable the IdP_client to securely interact with the IdP_server and store a user session token. Using a web portal made available by the IdP_server, the User generates an activation code. Then, using her mobile device, the User digits it into IdP_client.

SSO Phase. The actual execution of the native SSO flow is shown in Figure 7 and, being similar with the FB native SSO solution, is described in Appendix. The two differences are:

- the generation of the sig parameter in Step 5 of Figure 7. Instead of using a value that is the same for IdP_client installation, we use the activation_code value (different for each user);
- the token type of the token_C in Step 6 of Figure 7. Instead of using bearer tokens (the default tokens of OAuth), we use digitally signed tokens that contain claims about the user authentication phase (e.g., a JWT). We required at least three claims: an issuer identifier (IdP_id), a user identifier (u_id), the audience that the token is intended for (C_id).

Thus, compared to the FB solution, we propose the following changes:

- we require the described activation phase that replaces Steps 3-5 of FB native SSO solution (Figure 5);
- we use the activation_code to obtain the sig parameter;
- token_C contains claims about the user authentication phase and is digitally signed.

Note that, these changes are enough to mitigate the security vulnerabilities reported in Section 3.4, in detail:

- *Phishing*: is mitigated by the fact that users do not have to insert their IdP credentials on mobile native apps. Obviously, our solution cannot avoid the phishing if a malicious app shows a fake login form and the user decides to enter her IdP credentials. However, note that, even if an organization is able to develop a properly secured system, the phishing attack can be performed exploiting users lack of attention. Thus, at the basis of the phishing mitigation, there is the user education (e.g., do not download email attachments sent by people you do not know). In our solution, we will train our users to not enter their IdP credentials inside mobile apps. In this way, while in the FB solution the user (even a security expert) cannot detect the ongoing attack (a malicious C can create the same login form of FB), in our model if a malicious app asks to insert the IdP credentials, the user can detect the ongoing attack;
- *User Impersonation*: to avoid the user impersonation attack, we have decided to adopt the token type introduced by the OpenID Connect solution.[9] In this way, C does not need to ask for user information from the IdP_server and can directly check on the phone if the token was released to itself;
- *Client (IdP_client) Impersonation*: is mitigated as the IdP_client app is authenticated by the IdP_server using the sig parameter generating with an activation_code that is different for each user. By doing so, even if in principle, the activation_code of the IdP_client app can be accessed by the owner of the phone, she cannot

[9]We want to underline again, that as OAuth, also OpenID Connect is a browser-based solution and, thus, we cannot use it directly as it provides only a partial support for the mobile native SSO scenario.

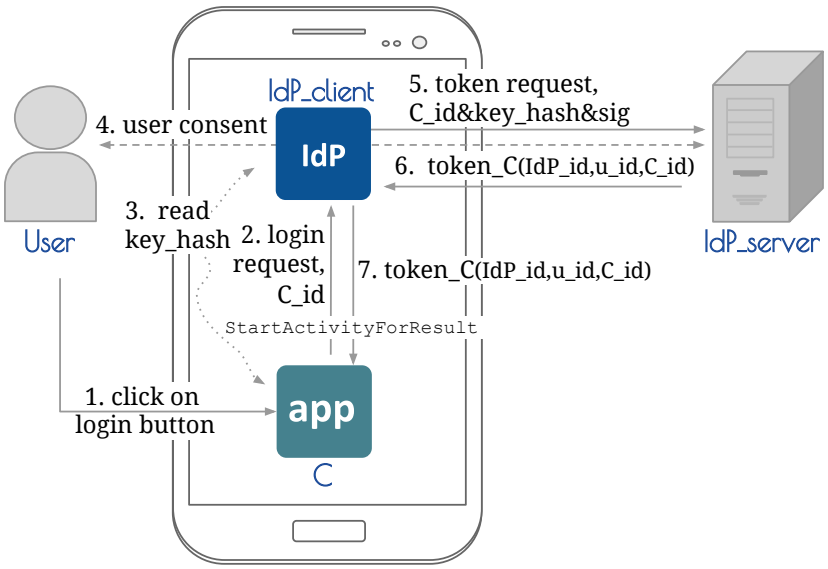


Figure 7: Our Native SSO Solution.

reuse it for obtaining token related to other users apart from herself.

We want to stress that the activation phase will be performed only the first time the user interacts with the IdP_client app. Thus, after the download and the activation of the IdP_client app, users can directly open the different C apps as usual. A scenario to be covered is the following: what would happen if a user starts C without having already activated the IdP_client? A possible solution is described here: if the IdP_client is installed on the phone, C will perform the login request to the IdP_client, and the IdP_client will show an alert saying that the user needs to complete the activation with the IdP_client app. Otherwise, C will show an error requiring the IdP_client installation.

5 RELATED WORK

A first attempt of native SSO standardization is carried out by the OpenID Foundation with the working group NAPPS (Native Applications) (OIDF, 2014b). Our model has some similarities with the NAPPS project, such as the use of digitally signed token, and the idea of introducing a new entity into the mobile phone that manages: *(i)* token exchanges, and *(ii)* user authentication and authorization processes. However being only a draft specification, NAPPS lacks a security analysis and many technical details. The NAPPS group is waiting to deliver a complete specification of its solution, as the mobile OSs (both iOS and Android) have been developing new features for supporting the native SSO (see (Madsen, 2015b; Madsen, 2015a)). They will introduce a new mechanism to implement browser-based protocol, which is a middle ground between the two browser solutions currently used (embedded and external browser). We will perform a further analysis as soon as details are available.

There are many studies in the literature, such as (Armando et al., 2008; Bansal et al., 2012; Sun and Beznosov, 2012; Wang et al., 2012; Zhou and Evans, 2014), which focus on the analysis of the different SSO implementations and the description of common vulnerabilities and attacks caused by incorrect implementation assumptions. These work, however, performed the analysis of the SSO for web applications, while the focus of our study is on native apps.

An in-depth analysis of OAuth in the mobile environment—underlining possible security problems and vulnerabilities—is performed by (Chen et al., 2014) and (Shehab and Mohsen, 2014). In (Chen et al., 2014), the two main problems addressed are: *(i)* the use of OAuth as an authentication protocol, and *(ii)* the lack of a mechanism to securely perform redirection in mobile platforms. The difference between (Chen et al., 2014) and our work is the goal that we want to achieve. (Chen et al., 2014) aims to be a warning about the need for a clearer OAuth guideline for mobile app developers, whereas we want to propose a reference model for native SSO. In the context of authorization, (Shehab and Mohsen, 2014) has a similar goal. They propose a new framework for securing the OAuth flow in smartphones. This framework requires the use of an embedded browser managed by a trusted app. In this way it does not require the download of an app for every different IdP and

provides the required separation between C and the UA. However, the level of detail provided is not sufficient to clarify, for example, how this solution prevents phishing and client impersonation attacks (described in Section 3.3).

About the Facebook solution for native apps, we based our study on the examples of attacks found in (Khorana, nd; Homakov, 2013; Goldshlager, 2013; Yao, 2010; Wulf, 2011).

6 CONCLUSIONS

The lack of security guidelines for mobile native SSO solutions has driven us to perform a detailed analysis of Facebook login current implementation. Starting from this analysis, we have contributed in this way: *(i)* we have extracted a rational reconstruction of the Facebook native SSO flow, *(ii)* we have performed a security analysis clarifying which are the security and trust assumptions and threat model for a native SSO scenario, and *(iii)* we have generalized the Facebook solution in a SSO model capable of mitigating the identified vulnerabilities.

As future work, we plan to *(i)* provide an implementation of our proposed solution, *(ii)* use a formal technique to provide a formal analysis of our solution, and *(iii)* refine our solution taking into account the authorization aspect.

ACKNOWLEDGEMENTS

This work has partially been supported by the Activity no. 16298, “Federated Identity Management System” (FIDES), funded by the EIT Digital.

REFERENCES

Armando, A., Carbone, R., Compagna, L., Cuéllar, J., and Tobarra, L. (2008). Formal Analysis of SAML 2.0 Web Browser Single Sign-On: Breaking the SAML-based Single Sign-On for Google Apps. In *Proceedings of the 6th ACM workshop on Formal methods in security engineering (FMSE ’08)*, pages 1–10.

Bansal, C., Bhargavan, K., and Maffeis, S. (2012). Discovering Concrete Attacks on Website Authorization by Formal Analysis. In *25th IEEE Computer Security Foundations Symposium (CSF’12)*, pages 247–262.

Boyd, R. (2012). Getting Started with OAuth 2.0. http://it-ebooks.info/read/664/.

Chen, E., Pei, Y., Chen, S., Tian, Y., Kotcher, R., and Tague, P. (2014). OAuth Demystified for Mobile Application Developers. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.

Chin, E., Felt, A. P., Greenwood, K., and Wagner, D. (2011). Analyzing Inter-application Communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, pages 239–252.

Facebook (2015). Getting Started with the Facebook SDK for Android. https://developers facebook. com/docs/android/getting-started/facebook-sdk-for-android/.

Facebook (2016). Signed Requests. https://developers. facebook.com/docs/reference/login/signed-request.

Goldshlager, N. (2013). How I hacked any Facebook Account...again! http://www.nirgoldshlager.com/2013/03/how-i-hacked-any-facebook-accountagain.html.

Homakov, E. (2013). How we hacked facebook with OAuth2 and Chrome bugs. http://homakov.blogspot.no/2013/02/hacking-face book-with-oauth2-and-chrome.html.

IETF (2012a). The OAuth 2.0 Authorization Framework. http://tools.ietf.org/html/rfc6749.

IETF (2012b). The OAuth 2.0 Authorization Framework: Bearer Token Usage. https://tools.ietf.org/html/rfc6750.

IETF (2015). JSON Web Token (JWT). https://tools.ietf.org/html/rfc7519.

Khorana, T. (n.d.). Fake Facebook Phishing Attack. https://sites.google.com/site/mobilesecuritylabware/9-mobile-phishing/post-lab-activities/lab-1-fake-facebook-phishing-attack.

Madsen, P. (2015a). Mobile OS Developments & Native Application Authentication. https://www.pingidentity.com/en/blog/2015/06/19/mobile_os_developments_native_application_authentication.html.

Madsen, P. (2015b). NAPPS has left the building (but is still on the front lawn). https://www.pingidentity.com/en/blog/2015/07/22/napps_has_left_the_building_but_is_still_on_the_front_lawn.html.

OASIS (2005). SAML V2.0 technical overview. https://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf.

OIDF (2014a). OpenID Connect Core 1.0. http://openid.net/specs/openid-connect-core-1_0.html.

OIDF (2014b). OpenID Connect Native Application Token Agent Core 1.0. http://openid.bitbucket.org/draft-native-application-agent-core-01.html.

Shehab, M. and Mohsen, F. (2014). Towards Enhancing the Security of OAuth Implementations in Smart Phones. In *IEEE International Conference on Mobile Services (MS)*, pages 39–46.

Sun, S. and Beznosov, K. (2012). The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS’12)*.

Wang, R., Chen, S., and Wang, X. (2012). Signing Me onto Your Accounts through Facebook and Google: A

Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services. In *IEEE Symposium on Security and Privacy (SP)*, pages 365–379.

Wang, R., Zhou, Y., Chen, S., Qadeer, S., Evans, D., and Gurevich, Y. (2013). Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization. In *Proceedings of the 22Nd USENIX Conference on Security (SEC'13)*, pages 399–414.

Wulf, A. (2011). Stealing Passwords is Easy in Native Mobile Apps Despite OAuth. http://welcome.totheinter.net/2011/01/12/.

Yao, Y. (2010). A serious OAuth security hole in Facebook SDK. http://security-n-tech.blogspot.it/2010/11/serious-oauth-security-hole-in-facebook.html.

Zhou, Y. and Evans, D. (2014). SSOScan: Automated Testing of Web Applications for Single Sign-On Vulnerabilities. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 495–510.

APPENDIX

For further information, we describe below the entire flow of our native SSO solution.

Registration Phase. We require a registration phase similar to the one of FB, where C developer has to enter the certificate fingerprint of C (key_hash) and obtains a C identifier (C_id). This is to avoid the *client impersonation* threat described in Section 3.3. In addition, in this phase, C and IdP_server have to agree on a signature algorithm and exchange the required public key.

Activation Phase. Using a web portal made available by the IdP_server, the User generates an activation code. Then, using her mobile device, the User digits it into IdP_client.

SSO Phase. The SSO flow, shown in Figure 7, is composed by the following steps: in Step 1, the user opens C and clicks the login button. When a user clicks on the login button an Activity from IdP_client using an explicit intent and the method `startActivityForResult` is invoked (Step 2). C is put on wait for the Activity to return a result. In Step 3, IdP_client read the certificate fingerprint (key_hash) of C, and in Step 4, presents a form that asks the user whether to authorize C to access her digital identity. In Step 5, IdP_client sends a request to IdP_server to get a fresh access token for C. The request contains C_id, key_hash and the sig parameter (a hash value obtained using the activation_code value). IdP_server checks whether the provided key_hash matches the one previously provided in the registration phase for the given C_id. If there is a mismatch then the flow is interrupted returning an error, otherwise the response (Step 6) contains a fresh token (token_C). In Step 7, IdP_client returns control to C and provides the token_C as result of the invoked Activity (see Step 2). C must validate the signature of the token according to the algorithm and using the public key specified in this registration phase. If the signature is valid, C has to perform two other steps: *(i)* verify if the issuer field correspond to the IdP_server, and *(ii)* check if the value of the audience field matches with the C_id. If the token is valid, C can extract the user information. This proves that C is under control of the corresponding user.