NTNU – Trondheim
Norwegian University of
Science and Technology

Authentication and Authorization for Native Mobile Applications using OAuth 2.0

Dag-Inge Aas

Master of Science in Computer Science
Submission date: June 2013
Supervisor: Tor Stålhane, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

NORWEGIAN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Abstract

Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Computer and Information Science

Master of Science in Computer Science

Authentication and authorization for native mobile applications using OAuth 2.0

by Dag-Inge Aas

OAuth 2.0 has in the recent years become the de-facto standard of doing API authorization and authentication on mobile devices. However, recent critics have claimed that OAuth does not provide sufficient security or ease-of-use for developers on mobile devices. In this thesis, I study four approaches to mobile authorization using OAuth 2.0, and suggest an improved solution based on current industry best-practices for security on Android. The end result is a solution which provides a native authorization flow for third-party developers to integrate with an existing API endpoint. However, the thesis shows that even with current industry best-practices the proposed solution does not provide a completely secure approach, and developers must keep the security consequences of that fact in mind when implementing OAuth on mobile devices.

NORGES TEKNISK-NATURVITENSKAPELIGE UNIVERSITET

Sammendrag

Fakultet for informasjonsteknologi, matematikk og elektroteknikk
Institutt for datateknikk og informasjonsvitenskap

Master i Teknologi

Authentication and authorization for native mobile applications using OAuth 2.0

av Dag-Inge Aas

OAuth 2.0 har de siste årene blitt industristandarden når man ønsker å gjøre API-autorisasjon og autentikasjon på mobile enheter. Likevel har OAuth blitt kritisert for å ikke tilby tilstrekkelig sikkerhet eller brukbarhet ved implementasjon for utviklere på mobile enheter. I denne avhandlingen skal jeg studere 4 måter å gjøre autorisasjon på mobile enheter med OAuth 2.0, og foreslå en forbedret løsning basert på industristandarder for sikkerhet på Android. Resultatet er en løsning som tilbyr en innebygd løsning for tredjepartsutviklere som ønsker å integrere sin applikasjon mot et eksisterende API. Likevel viser løsningen at selv med dagens industristandarder for sikkerhet så kan ikke den foreslåtte løsningen garantere en sikker løsningen, og at utviklere derfor må ta med konsekvensene av dette i betraktningen når de skal implementere OAuth på mobile enheter.

Acknowledgements

I would like to thank Tor Stålhane, Magne Mæhre and Jeanine Lilleng for their invaluable assistance in supervising this thesis. I would also like to thank Telenor Comoyo that initiated this thesis, and accepted me as their master student.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Authorization and authentication on mobile devices

Many applications require a user to to provide credentials to authenticate the user. While commonplace, such a solution requires these credentials to be sent with every request to the server, which is, even on encrypted channels, not ideal. Replacing the users main credentials with application specific, time- and scope-limited tokens is a much better approach to user authentication to protected API calls. The purpose of authorization frameworks such as OAuth is to replace the need for the user's main credentials, and create a documented flow for requesting such tokens with the user's consent, all while protecting the user's credentials from falling into the wrong hands.

There are, however, some challenges when faced with OAuth 2.0 on mobile devices. Mobile devices are regarded as public clients, unable to protect the client credentials from a possible attacker. However, in some cases, it is desirable to use OAuth on a mobile device, both in a native way without using integrated browsers, and in a secure way. This thesis will focus on proposing both a native, and secure solution to the problem of OAuth on mobile devices, in addition to discussing the mitigations and risk factors of using OAuth on a mobile device, both from a user's and from a developer's perspective.

1.2 Purpose statement

OAuth 2.0 is an authorization framework commonly used to grant applications limited access to a user's resources without exposing the users credentials to the application. Many applications are currently using OAuth 2.0 to authorize access to a user's Facebook or Google information through either mobile or web applications. However, the current OAuth 2.0 specification does not support native mobile applications where opening a browser is not possible or using one to do authorization is unwanted. This thesis seeks a solution to authentication and authorization using the OAuth 2.0 framework on native mobile devices, without leveraging the browser.

The main issue with mobile authentication and authorization using OAuth 2.0 is the storing of client_id and client_secret. An API provider might wish to limit API access to only approved clients, and a developer would want to protect his clients' credentials to avoid misuse of such credentials. An API provider must also be able to revoke such credentials should they be misused, without the risk of breaking existing clients. This is especially a problem with mobile applications where distribution of new credentials are not instant, and some application marketplaces require approval taking several days before new credentials can be distributed to users. In addition, we do not want to use browser-based views in a native application as it has been shown to be vulnerable to attack from a malicious application [3].

In either case, it is necessary to create a framework for authentication and authorization on mobile devices using native input, and standard HTTP calls, preferably against a REST-based API interface.

1.3 Research Outcomes

The desired outcome of this thesis is to present a viable solution to secure authorization using OAuth 2.0 on native mobile devices. By secure I mean avoid leaking client credentials if possible, and above all, secure the user's master credentials from misuse by mischievous applications. By native I mean to avoid, if possible, the use of integrated browser-based solutions in the application themselves.

The thesis will focus on the security aspect of OAuth 2.0 on mobile devices, identify which solutions are deployed today, and propose possible new ways to do authorization using OAuth 2.0.

1.4 Research Questions

I have identified the following Research Questions which will be answered by this thesis.

- **RQ1:** What methods are used today for authentication and authorization on mobile devices leveraging OAuth 2.0?
- **RQ2:** Are there known attacks or vulnerabilities in today's solutions, and how are they executed?
- **RQ3:** How can we, or can we, implement an authorization flow without using browser-specific solutions, such as WebView on Android?
- **RQ4:** How can we, or can we, securely store client_id and client_secret in applications distributed through Google Play (Android Market)?
- **RQ5:** How can we improve the solutions to authorization on mobile devices that exists today?

1.5 Research Method

My research method will largely focus on a structured literature review (SLR) of recent research in mobile authentication and authorization, together with rapid prototyping of viable solutions identified in the SLR. I will also draw heavily on the preliminary study into authorization solutions for the web in the project report "Authorization Solutions on the Internet" [4]. In addition to the literature review of academic resources, I have chosen to study four methods of using OAuth on mobile devices, `WebView`, Resource Owner Password Credentials Grant, Facebook's SDK and Google's SDK. I will discuss what these solutions entail in the relevant sections in Chapter 4.

1.6 Test environment

I have selected the Android Operating System [5] as the test environment, because it is UNIX-based, open source and free to use. It is also one of the leading smart phone operating systems on the market. All code examples are written to work on Android 2.3.3 (API version 10) or later. However, the best practices used are based on recommendations from the Android Developer Community and documentation for version 10, which is subject to change. API level 10 was selected primarily because, at the time of writing, it supports 94.1 percent of the Android smartphone market [2]. See also Table 1.1.

Version	API level	Distribution
1.6	4	0.1%
2.1	7	1.7%
2.2	8	4.0%
2.3 - 2.3.2	9	0.1%
2.3.3 - 2.3.7	10	39.7%
3.2	13	0.2%
4.0.3 - 4.0.4	15	29.3%
4.1.x	16	23.0%
4.2.x	17	2.0%

Table 1.1: Android Platform Version Distribution in market as of 2013-04-02 [2]

The code in this thesis was tested on both the Android Emulator running Android Version 2.3.3, and a Nexus 4 running Android 4.2.2. See Table 1.2 for more information on the exact software version running on the test devices.

The code examples do not adhere to any particular coding standard, nor are they recommended for production use. They are written to be as legible as possible. Some code may be omitted for the sake of brevity.

	Android Emulator	LG Nexus 4
Android Version	2.3.3	4.2.2
API Version	10	17
Baseband Version	N/A	M9615A-CEFWMAZM-2.0.1700.48
Kernel Version	2.6.29-00261-g0097074-dirty	3.4.0-perf-g7ce11cd
Build Number	sdk-eng 2.3.3 GRI34 101070 test-keys	JDQ39

TABLE 1.2: Devices used to test the implementations of the proposed solutions

1.7 Report Outline

The report is structured as follows. First, I give an introduction to what authorization frameworks and OAuth is, and what challenges we face with regards to mobile devices and applications. Second, I study the Android Security Model, to better understand the security principles in the Android platform. Third, I do a Structured Literature Review of relevant research in academia on OAuth security on mobile devices. I then examine four approaches identified through the SLR and through popular implementations from API providers. These are evaluated based on security. Based on the findings of this study, I implement and discuss best-practices for a proposed solution to native authorization flows with OAuth 2.0. Finally, I discuss the research questions and conclude my thesis.

Chapter 2

Background and Related Work

2.1 Authorization Frameworks

Authorization frameworks create a documented flow to enable users to grant third-party applications access to their web resources without sharing their login credentials or the full extent of their data. These flows will often issue the application a special key that grants the application access to a limited subset of a user's protected resources.

2.1.1 Conclusion from Authorization Framework Study

In the preliminary project report for this thesis, I studied four authorization frameworks and protocols to determine the best authorization framework currently in widespread use. This study concluded that OAuth 2.0 is the most used, versatile and secure solution of the frameworks studied.

2.1.2 History

Both Google, Flickr, Yahoo and other major API endpoints started out with their own proprietary solutions to this problem. While all of them solved the same fundamental problem, there were a few different solutions, and none of them were interoperable. Each time a developer wanted to integrate with a new service, it meant learning a new framework and writing custom code to handle it. This lead to a kind of lock-in, and a fragmented space meant that fewer client libraries existed, making it harder to integrate with a small, custom service.

Several efforts to standardize a way of doing authorization on the web were started, but the one that had the most success was OAuth 1.0. OAuth 1.0 started from a need to allow Dashboard Widgets access to a users content. Blaine Cook, Chris Messina and Larry Halff teamed up with David Recordon, to create the OAuth Discussion Group in April 2007, with the purpose of creating an open standard for authorization on the web. The specification was published as

RFC 5849 in April 2010 [6]. This later lead to widespread adoption, and Open APIs became an important part of the Web 2.0 movement, especially for the social networks.

However, OAuth 1.0 was only the beginning, and lacked a standardized way of performing authorization negotiation for javascript-based web applications which became more and more popular. In addition, OAuth 1.0 required a custom encryption implementation to identify the client with the server, making implementation tedious and error prone. Because of this, the work on an alternative, OAuth WRAP, which was later renamed OAuth 2.0, began.

2.2 OAuth 2.0

OAuth 2.0 is the sequel to the popular, industry standard, OAuth 1.0(a). It is not backwards compatible, but retains much of the established approach and architecture of OAuth 1.0. The goal of OAuth 2.0 is to "enable a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf."[7, abstract]

OAuth 2.0 started as a collaboration between Microsoft, Google, Yahoo and Salesforce.com to address the issues in OAuth 1.0. Although not originally intended as a new version of OAuth, the protocol was contributed to the IETF OAuth working group in 2009. In 2011, Facebook announced OAuth 2.0 as part of the API roadmap [8], becoming one of the world's biggest API providers using OAuth 2.0.

Since its release, the framework has gained popularity amongst the worlds biggest API providers, including Facebook, Google, Github, Twitter and Microsoft. While there are few complete open source implementations of the specification, an ongoing project under Apache, called Amber, is actively working on a full implementation.

2.2.1 Flows

Flows in OAuth denote the ways a client can obtain an access token from the authorization server. While an implementation is not required to support all flows, the implementation is required to support at least one OAuth authorization flow to be called OAuth compliant. These flows are, however, quite loosely defined, and one implementation of any flow might differ slightly from another implementation. This leads to interoperability issues between implementations, which makes it harder to create general client libraries for authorizing with OAuth. Below, we will discuss the actors, and the possible flows between them, starting with a generalized overview of the authorization flow.

Figure 2.1 shows the interaction between Client, Resource Owner, Authorization Server and Resource Owner as specified in RFC6749 [7, Section 1.2]. The purpose of this flow is to obtain an access token to act on behalf of the resource owner on the resource owners protected resources.

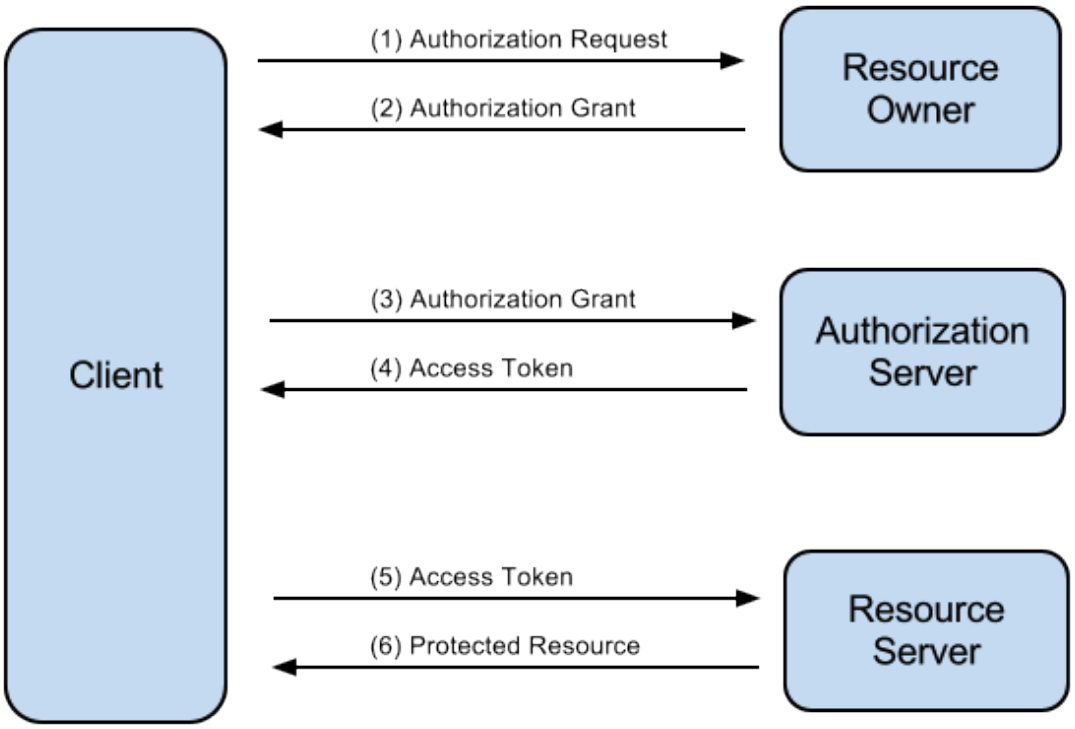

FIGURE 2.1: OAuth 2.0 Generalized Authorization flow.

1. The client requests authorization from the resource owner. The authorization request is mostly done indirectly through the Authorization Server, but can also be requested directly from the Resource Owner.
2. The client receives an authorization grant, which is a credential representing a resource owner's consent to authorization.
3. The client requests an access token by authenticating with the authorization server and presenting the authorization grant.
4. The authorization server authenticates the client and validates the authorization grant and, if valid, issues an access token.
5. The client requests the protected resource from the resource server and authenticates by presenting the access token.
6. The resource server validates the token and, if valid, serves the request.

It is important to note that the validation in the last step is not defined by RFC6749, and the implementation details are left to the implementor.

OAuth supports several grant flows to enable a client to procure an access token. These are examined below.

2.2.1.1 Authorization Code Grant

The authorization code grant type is used to obtain both access tokens and refresh tokens and is optimized for confidential clients. Since this is a redirection-based flow, the client must be able to interact with the resource owner's user-agent (typically a web browser) and be able to receive incoming requests (via redirection) from the authorization server. [7, Section 4.1]

For an explanation of the actors in Figure 2.2, see Appendix A.1.

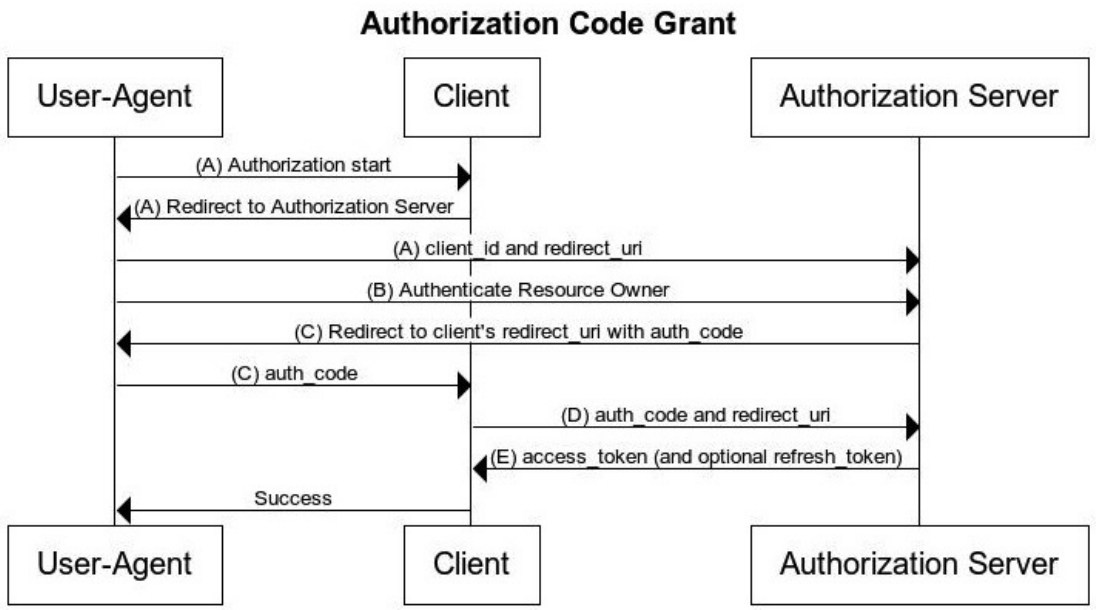

FIGURE 2.2: OAuth 2.0 Authorization Code Grant flow

A. The client initiates the authorization request by redirecting the user-agent to the authorization server. This is done through a specially crafted URL containing the client id and redirect URL as URL parameters.

B. The authorization server authenticates the resource owner. How this is done is beyond the scope of RFC 6749. The authorization server also verifies that the resource owner either grants or denies access to the client for the given scope.

C. The authorization server redirects the user-agent back to the client using the redirect URL given in step A. It also adds an authorization code as a parameter.

D. The client requests an access token from the authorization server by presenting the authorization code and by authenticating itself.

E. The authorization server authenticates the client and validates the authorization code. An access token is returned in a JSON encoded format.

2.2.1.2 Implicit Grant

The implicit grant type is used to obtain access tokens (it does not support the issuance of refresh tokens) and is optimized for public clients known to operate a particular redirection URL. These

clients are typically implemented in a browser using a scripting language such as JavaScript. [7, Section 4.2]

Unlike the authorization code grant type, in which the client makes separate requests for authorization and for an access token, the client receives the access token as the result of the authorization request.

For an explanation of the actors in Figure 2.3, see Appendix A.1.

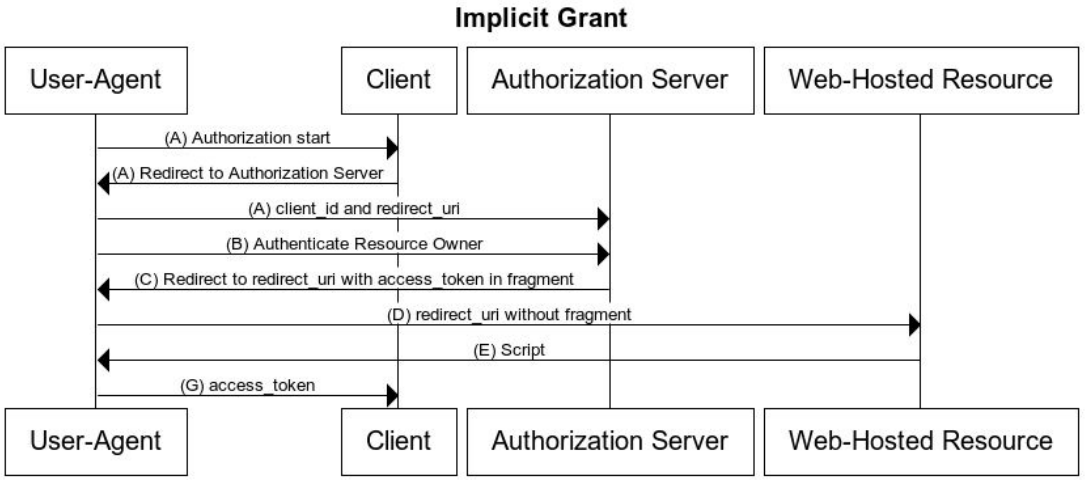

FIGURE 2.3: OAuth 2.0 Implicit Grant flow

A. The client initiates the authorization request by redirecting the user-agent to the authorization server. This is done through a specially crafted URL containing the client id and redirect URL as URL parameters.

B. The authorization server authenticates the resource owner. How this is done is beyond the scope of RFC 6749. The authorization server also verifies whether the resource owner grants or denies access to the client for the given scope.

C. The authorization server redirects the user-agent back to the client using the redirect URL given in step A. It also adds the access code as a URL fragment.

D. The user-agent makes a request to the web-hosted resource without the fragment, and stores the fragment locally.

E. The web-hosted resource returns a web-page containing a script used for extracting the access token from the user-agent.

F. (Not shown) The user-agent runs the script and extracts the access token from the URL fragment.

G. The user-agent passes the access token to the client.

2.2.1.3 Resource Owner Password Credentials Grant

The resource owner password credentials grant type is suitable in cases where the resource owner has a trust relationship with the client, such as the device operating system or a highly privileged application. This flow is mainly meant for migration purposes. [7, Section 4.3]

For an explanation of the actors in Figure 2.4, see Appendix A.1.

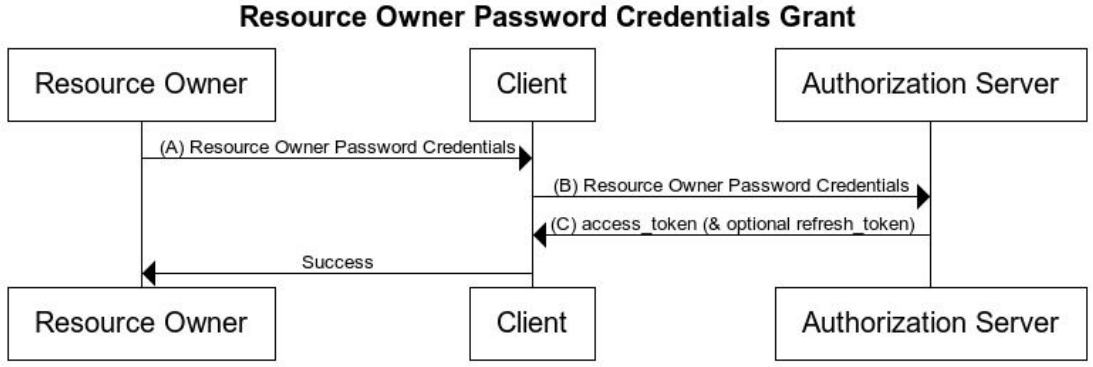

FIGURE 2.4: OAuth 2.0 Resource Owner Password Credentials Grant flow

A. The Resource Owner submits his/her password credentials to the client.

B. The client submits the resource owner's password credentials to the authorization server.

C. The authorization server authenticates the client and validates the credentials, and returns an access token to the client. The client must delete the resource owners credentials immediately upon receiving the access token.

2.2.1.4 Client Credentials Grant

The client can request an access token using only its client credentials (or other supported means of authentication) when the client is requesting access to the protected resources under its control, or those of another resource owner that have been previously arranged with the authorization server. [7, Section 4.4]

For an explanation of the actors in Figure 2.5, see Appendix A.1.

A. The client authenticates with the authorization server and requests an access token from the token endpoint.

B. The authorization server authenticates the client, and issues an access token.

2.2.2 Authorization Frameworks as Identity Management

One of the biggest applications of OAuth 2.0 today is identity management [9]. "Sign in with Facebook" or "Sign in with Google" is scattered across the web in login screens, even completely

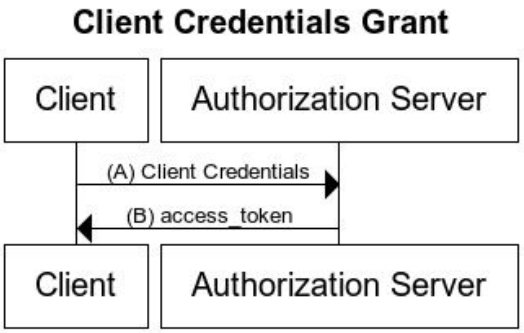

FIGURE 2.5: OAuth 2.0 Client Credentials Grant flow

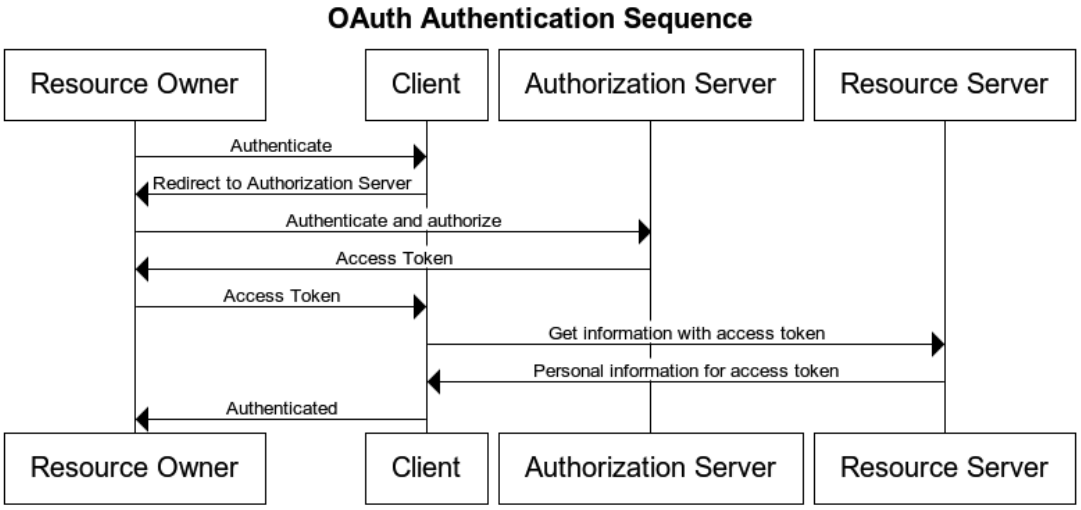

FIGURE 2.6: OAuth 2 Pseudo-Authentication Example

replacing the regular username/password combination on some sites. Some designers predict that identity management systems will spread due to adoption by large, established user bases on applications such as social networking platforms [9]. The widespread use of Facebook's and Google's APIs, in addition to the simplicity in implementing an OAuth 2.0 client, has driven the adoption of the OAuth 2.0 Authorization Framework. Due to the relationship of trust between these resource providers and clients, the APIs have also come to support pseudo-authentication. Figure 2.6 shows an example of how such pseudo-authentication is achieved through the use of states and personal information stored for the resource owner.

In OAuth 2.0, an access token is unique to a each resource owner and client. A resource provider must therefore validate the identity of the resource owner before issuing an access token to a client. Due to the client trusting the resource provider (RP) of doing this correctly, the client can, through the use of the RP's API, figure out the identity of the resource owner based on the access token. The access token can therefore become not only a key for the RP to know what a client can access, but for a client to know who the resource owner is. See also Figure 2.6 for an example flow to support OAuth 2.0 pseudo-authentication.

2.2.3 Challenges

Even though OAuth 2.0 has been established as an industry standard with 6 out of 7 of the industry leading API providers using OAuth 2.0[4], the framework is wrought with problems. Eran D. Hammer, former chief editor of the OAuth 2.0 specification, has publicly criticized the framework for its inherent interoperability and security problems [10]. Hammer also identified that one of the biggest challenges with OAuth 2.0 is the lack of support for native mobile application support, with applications having to rely on opening browser-based views in their applications to let the user grant access to his or her protected resources [11].

2.2.3.1 Native Mobile Device Applications

The challenge OAuth faces on mobile devices relates to the authorization process. OAuth is a browser-based authorization solution, meaning that for an application to securely obtain an access token, an application must leverage a browser to support the flow. This is often unwanted by application developers for mobile devices, as opening a browser-session from your application not only removes the user from the context of the application, possibly hurting usability, but can also be costly computing-wise with regards to e.g. memory management. However, we still see many mobile applications utilizing amongst others Facebook Connect and the Facebook Graph API, which uses OAuth 2.0, to create rich experiences for their users.

Facebook solves this problem of removing the user from the application context by using a redirect-based flow native to the Android environment, by requiring that the Facebook for Android application is installed on the client device, and using built-in GUI components in a familiar environment to retrieve approval from the user. See also Section 4.2 for an example implementation of this flow. This solution is becoming more and more prevalent in the mobile space for OAuth.

2.2.3.2 Keeping secrets secret on mobile devices

One of the major problems today with OAuth on mobile devices is keeping client credentials secret. This is needed so you can separate one application from the other, revoke a specific application, or grant an application special privileges. Twitter recently introduced rate-limiting on a client basis [12]. For developers, this meant that they could not deploy millions of applications using one set of client credentials, for example to create their own Twitter client. This meant that many existing client developers had to pull back their application from the application stores, and some even lost their business. However, one Twitter client used the API without rate-limiting, namely Twitter's own client, which is distributed through the same mobile application stores. In February 2013, hackers successfully extracted all client credentials from all of Twitter's official clients, including both Android and iOS applications [13]. These were posted publicly, and because of the distributed nature of requests, and the fact that Twitter could not just revoke the credentials, this meant that developers now had free reign to use Twitter's own client credentials to avoid the rate-limiting set on the API.

While the OAuth specification clearly states that a mobile device is a public client, meaning that the device is incapable of protecting its own secrets, it is still a valid requirement that some mobile applications have special privileges, and that the server can be certain of which application is making the request, so that the resource owner can safely revoke access to a mischievous application, without disrupting other applications.

2.3 Android Security

This thesis will look at several of the security aspects of Android applications. It is therefore necessary to have a basic understanding of the security context that Android applications run in, and the consequences of this.

2.3.1 Android Architecture

The Android platform consists of a stack with several layers running on top of each other, where lower-level layers provide services to upper-level layers. For a graphic overview of the Android platform, see Figure 2.7.

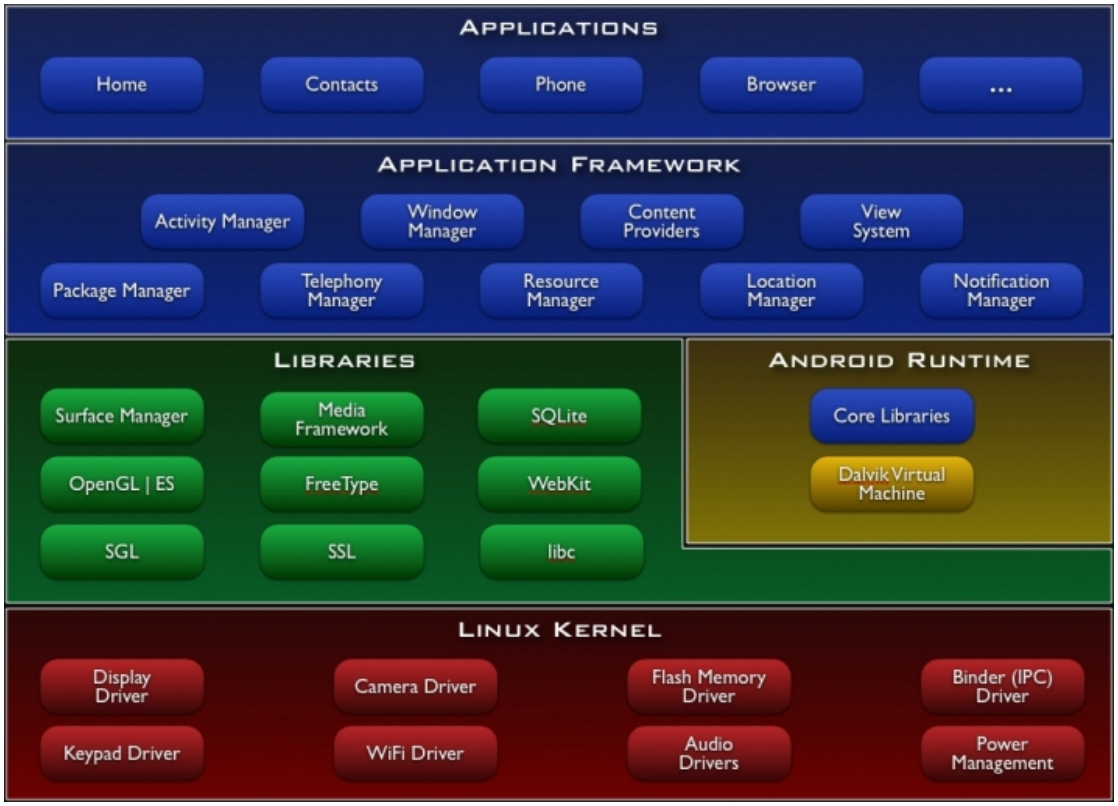

FIGURE 2.7: Android Platform Overview [1]

At the bottom layer, we find the Linux kernel. The Linux kernel provides hardware abstractions, a common interface for drivers to integrate with, and enforces some basic separation between applications through basic security principles inherit in any Linux system.

Above the kernel, we find the native libraries. These provide common services for other applications, such as the Surface Manager, which is responsible for the graphics on the device's screen. In addition we find graphics libraries, browser rendering engine, SQLite and so on. These libraries run as processes on the underlying Linux kernel. At this level, we also find the Android runtimes. Each application runs in its own instance of the Android runtime, and in each runtime is an instance of the Dalvik Virtual Machine, a mobile-optimised virtual machine. Each of these instances runs the Android core libraries, such as I/O, class libraries and so forth.

Above the Android runtime is the application framework. These are processes running on the Dalvik VM that provides services to the applications. This can be the Window Manager, Notification Manager, or many more. Anyone can write code that runs within the application framework, but the framework does not run directly as a process on the Linux kernel. This provides an important security abstraction.

Finally, applications run on the top layer. There is no difference between applications that "regular" developers write, and the applications that Google writes. They all run in the same context, with the same APIs. Applications can communicate using Receivers, Services and Content Providers, and the challenge in securing an application lies in these methods of communicating, rather than access through low-level architecture.

2.3.2 Android Security Model

The Android Security Model is built upon the foundations of the Linux Security Model. The Linux Security Model is built around the concepts of users and groups. Each user is assigned a user ID (UID), and each group is assigned a group ID (GID). These are unique, and used to differentiate one user from another. Each group can have multiple users, and each user can belong to multiple groups. Every resource in the Linux environment is usually a file. Each file has a set of permissions: user, group and world. Each set of permission can include read, write or execute permissions, specifying which operations are allowed on a given resource for a given user or group.

When an Android package is installed, a new UID is created, and the application will run under this UID. This is different from systems such as Windows, where each application runs in the user's context, and with the current running user's permissions. All data stored by the application is stored with the same UID, and special permissions set by the developer denote the access for group and world. The standard and best-practice for e.g. user credentials uses MODE_PRIVATE, which means that the resulting file can only be read by the current application, and other applications cannot access this file without access to root. Each UID is device specific, meaning that an instance of the application running on another device will not necessarily have the same UID. This means that every file created by an application which is only readable for the owner, can only be seen by the owner, and other applications cannot access it without superuser privileges.

It is also worth noting that Android applications are capable of running native code, which runs outside the DalvikVM. However, inclusion of native code in an Android application does

not alter the security model, and the same architectural separations between applications is enforced, regardless of the type of application (native or Dalvik, or a mix of the two).

2.3.3 Android Application Signing

To increase the trust between developer and user, all Android applications must be digitally signed by the developer prior to release. This digital signature is used to identify the developer of the application. A digital signature can only be generated using a digital certificate with an associated private key. Only people with access to this private key can create signatures, and it is therefore important to keep the secrecy and integrity of this private key.

Digital certificates and the associated private key can either be issued by a Certificate Authority, a trusted third-party that will make you prove that you are the developer in question before issuing a certificate, or they can be so called self-signed certificates. Self-signed certificates are the standard way of doing release signing in Android, and cannot to any degree of confidence tell the identity of the person signing the application, but a user can verify that the signature came from the same developer when comparing two different applications. This is usually solved by Google identifying a developer when the developer registers to have the application distributed through the Play Store (Android Market), therefore establishing trust between the user and developer.

2.3.4 Dealing with root access

A major caveat with Android security for application developers is the ability to root the device. Rooting a device means that the application has the capability to act as the superuser on the system. This is analogous to the Linux world where the superuser can access all files regardless of permission set on them. However, devices must be explicitly rooted by the user, unless an exploit is uncovered to escalate a given application's rights to superuser privileges.

The risk of rooting is therefore minimal as a regular user is unlikely to root her phone unless explicitly knowing about the dangers. We can therefore assume that secrets stored by the device on behalf of the user, such as an access token or refresh token, is stored securely. For a developer wanting to store secrets away from the user, the risk of rooting is a bigger threat. A malicious user can extract secrets from the application by rooting the device, and reading any file stored on the system. While it is possible to mitigate this threat through encryption methods, they are not relevant for most applications.

2.4 Related Work

I have, in preparation for this thesis, written a project report for Telenor Comoyo, "Authorization Solutions on the Internet" [4]. I will base my thesis on this work, and some of the sources uncovered during the research for that paper.

Authorization Frameworks, and OAuth in particular, have been a topic for research during the recent years, especially with regards to the security aspect of such frameworks. Even though few studies have touched upon a native mobile-centric approach to OAuth, some studies such as "Authentication and Authorization on Mobile Devices" [14], a bachelor thesis from the University of Gothenburg, have evaluated several frameworks for authentication and authorization on mobile devices. A preliminary search for related work did not uncover any relevant literature regarding proposals for authorization flows for mobile devices paired with OAuth 2.0, or any models that can guarantee a secure, usable or widespread approach to mobile device authorization. To more formally uncover relevant literature we will perform a Structured Literature Review (SLR) in Chapter 3.

There are also numerous developer guides to OAuth and the Facebook API which are well-written, and which will be the basis for today's solution on authorization on native mobile devices.

Chapter 3

Structured Literature Review

3.1 Introduction

A review of prior, relevant literature is an essential feature to any academic project. An effective review creates a firm foundation for advancing knowledge [15]. The purpose of this literature review is to uncover relevant literature related to secure authentication and authorization on native mobile devices, and to uncover proposed models for secure authentication that I can build upon in this thesis.

A Structured Literature Review (SLR) is a formal way of synthesising the information available from available primary studies relevant to a set of research questions. The use of structured literature reviews have traditionally been widespread primarily in medicine, but have in recent years become a more prevalent research method also in Computer Science [16].

Structured Literature Reviews in computer science stand apart from the more unsystematic surveys by using a strict methodological framework with a set of well defined steps carried out in accordance with a predefined protocol [16].

This review is based upon the approach set forth by Webster and Watson in "Analyzing the Past to Prepare for the Future: Writing a Literature Review" [15]. In addition, I will follow the guidelines in "How to do a Structured Literature Review in Computer Science" by Anders Kofod-Petersen [16]. The steps below are adapted from the approach outlined in these two papers.

3.2 Identification of the need for review

To identify similar work, I have performed a preliminary study of relevant work using an unsystematic approach. This has been done through the search venue "Google Scholar" using the search string: "OAuth secure native mobile authentication authorization" or variations of this.

This uncovered several relevant papers in OAuth and security, but few proposed models for authorization and authentication using a native mobile approach. The studies uncovered do not answer the research questions in Section 1.4.

3.3 Research Questions

The research questions are formulated in Section 1.4. Table 3.1 shows the research questions together with a short description.

Research Question	**Description**
RQ1 What methods are used today for authentication and authorization on -mobile devices leveraging OAuth 2.0?	There are several methods of doing authentication and authorization using OAuth 2.0 on mobile devices today. I wish to identify the different methods used, their advantages, and the drawbacks these present.
RQ2 Are there known attacks or vulnerabilities in today's solutions, and how are they executed?	I wish to uncover if there exists known attacks against existing solutions identified in RQ1, what threat they pose, and how to mitigate them, if possible.
RQ3 How can we, or can we, implement an authorization flow without using browser-specific solutions, such as WebView on Android?	One of the biggest gripes that developers have with OAuth-based authorization flows are the requirement on browser-based approaches on limited hardware such as smartphones. I wish to investigate if there are existing solutions that can be used without integrated browsers, or if I can propose a solution that does not require a browser, but still maintains the security of user's data.
RQ4 How can we, or can we, securely store client_id and client_secret in applications distributed through the iOS App Store or Google Play?	Are there ways of keeping client credentials safe and secret on a mobile application, even when distributing the applications through open systems such as the mobile application stores?
RQ4 How can we improve the solutions to authorization that exists today?	Considering the solutions that were uncovered in RQ1, how can we improve these solutions, or are there different and better solutions uncovered that should be regarded as best-practice?

TABLE 3.1: Research questions together with a short description

3.4 Search Strategy

The purpose of a documented search strategy is to have a set of steps, thereby making the work reproducible [16]. Below, I have documented the keywords and search venues used to collect primary studies.

3.4.1 Keywords

The keywords used in the SLR were identified from trial searches. These trial searches were performed on Google Scholar, as Google indexes several search venues and sorts them based on keyword relevancy, and I assume that the keywords identified will also yield the maximum amount of relevant research in other search venues as well. The keywords I have chosen are listed in Table 3.2

	Group 1	Group 2	Group 3	Group 4
Term 1	OAuth	authentication	native	mobile
Term 2		authorization	secure	

TABLE 3.2: Search keywords for Structured Literature Review

The search string will be implemented according to this formula. G# stands for Group [1,2,3,4] and T# stands for Term [1,2].

$$A = ([G1, T1])$$

$$B = ([G2, T1] \vee [G2, T2])$$

$$C = ([G3, T1] \vee [G3, T2])$$

$$D = ([G4, T1])$$

$$A \wedge B \wedge C \wedge D$$

3.4.2 Search Venues

I have chosen IEEE Explore and Google Scholar as my search venues. While Google Scholar may be seen as a controversial choice by many, studies show that Google Scholar is 17.6 percent more scholarly than materials found only in library databases [17], and that Google Scholar indexes 90 percent of engineering articles published after 1990 [18]. Since my research is limited to articles published in 2010 or later, we can assume that Google Scholar will give me the breadth needed for the SLR.

Since Google Scholar sorts papers according to relevancy, the results may vary. However, many other library indexers have recently done the same [17], so any search results is subject to change. To mitigate this, I also include IEEE Explore in my review, which can sort based on multiple metrics.

3.5 Inclusion and Exclusion criteria

To select the primary papers to study I have performed some pre-filtering on the results. The following pre-filters were applied to the results from the search venues.

- Remove duplicates, keep version from the most reputable source, or newest revision if applicable.
- Exclude studies published before 2010 and until, but not including, March 2013.
- Remove non-English articles

The relatively short timeframe for valid studies is needed since OAuth 2.0 was first introduced in May 2010. However, if any older, applicable studies for keeping secrets safe on client devices are found in the filtering process, it will be further studied. The preliminary study revealed no relevant papers or articles before 2010.

After the pre-filter, I ran each of the pre-filtered studies through some inclusion criteria to choose the papers that will be selected for review. This is done to ensure relevancy to the research questions. The relevant inclusion criteria (IC) can be seen in Table 3.3. The inclusion criteria are in the binary YES/NO-form. If a paper answers YES to any on the inclusion criteria, it is included for review. Articles that do not have a clear statement of purpose/goal will be excluded as they cannot be used as a basis for technology adoption. Furthermore, I have only included peer-reviewed articles in the study, but the final bibliography will include articles posted online from reputable sources as well. These will not be part of the Structured Literature Review, and will only be used to reflect the point of a single author, and not academia as a whole.

Inclusion Criteria	**Description**
IC1 The studies main concern is OAuth 2.0 on mobile devices	The study revolves around OAuth 2.0 on mobile devices, from any standpoint.
IC2 The study's main concern is OAuth 2.0 on mobile devices from a security standpoint	The study revolves around OAuth 2.0 on mobile devices, from a security standpoint.
IC3 The study's main concern is implementing a native authorization flow for OAuth 2.0 on mobile devices	The study revolves around finding a solution to native authorization flows, without leveraging a browser, on mobile devices.
IC4 The study's main concern is implementing a native authorization flow for OAuth 2.0 on mobile devices	The study revolves around finding a solution to native authorization flows, without leveraging a browser, on mobile devices.

Table 3.3: Inclusion Criteria for Structured Literature Review

3.6 Conducting the review

Conducting the review consist of searching the selected venues using the search string. While IEEE Explore supported searching for complex strings, Google Scholar seemed to create its own interpretation of the search string, and thus yielded many irrelevant results. Because of this,

only the three first pages, or 30 results, were included in the review. This number was chosen based on random sampling on pages beyond that, which showed only irrelevant results.

Google Scholar yielded 198 results. I also added an additional constraint to the search parameter to avoid results from IETF (-ietf). This was done to ensure that mostly peer-reviewed articles were returned, and not the specifications themselves. I only chose to examine 30 of the first results, due to reasons stated above. Of these 30 results, only four were deemed fit for further review.

IEEE Explore yielded 105 results. However, surprisingly enough, none of the articles returned passed the inclusion criteria. The results yielded from Google Scholar did not contain any relevant articles from IEEE either, leading me to believe that no articles regarding native mobile authorization with OAuth has been published as of March 2013 by IEEE.

#	**Title**	**Year**	**Reference**
1	POAuth: privacy-aware open authorization for native apps on smartphone platforms	2012	[19]
2	Method and system for conducting a monetary transaction using a mobile communication device	2012	[20]
3	Identity, Access Management and Single Sign-On Web-based Solutions	2012	[21]
4	Android Environment Security	2012	[22]

Table 3.4: Papers identified for further review

3.7 Results and discussion

Four papers were reviewed. All of these were published in 2012, which shows that the field of study is quite young, and we can expect an increase of literature in the coming years. It was quite disappointing to find so little literature, even though Google Scholar indexes several of the most popular engineering digital libraries such as IEEE Explore and ACM. This may be due to the relatively young field of research, since the specification itself is relatively young.

All four papers touched on the subject of security with regards to mobile devices and authorization. However, only one mentioned OAuth 2.0 specifically.

While I found some papers, they do not nearly cover everything I need for this thesis. I will therefore also include primary sources such as the specifications themselves and discussions from discussion boards and blogs in the study. Many prominent white hat hackers, such as Egor Homakov and Nir Goldshlager have studied the OAuth implementations of, amongst others, Facebook and Twitter, and have published a lot of articles regarding their methods.

In addition to this, I will also include some reference implementations from known, large API providers identified in [4]. The providers chosen are Facebook and Google, as they have the most up-to-date implementations of their Mobile SDK.

Chapter 4

Current Solutions

4.1 Introduction

We will look at today's solutions for authorization with OAuth 2.0 on native mobile devices. While our Structured Literature Research did not uncover any relevant papers to today's solution in academia, we can use Facebook and Google's implementations and recommendations to gauge what the best practices are in the industry today. There are also numerous blog posts and online articles studying the security of these APIs. Facebook and Google were chosen because they are the world's largest API providers, and they have the longest-running implementation of OAuth 2.0 [4]. Google has also recently released a new approach to OAuth 2.0 on mobile devices through the Google Play Services.

In this chapter we will look at four solutions, ranging from the very basic WebView flow, to the more native approaches that Google and Facebook uses. We will discuss their pros and cons, and conclude on what today's best practice is in regards to mobile authorization.

4.1.1 Shared code

The following code is assumed to be present when the examples are run. These are common for all proposed solutions, as they are not a direct part of the Android Standard Library, but are frequently used in the examples.

```
/**
 * Turn an InputStream into a String
 */
public String stringify(InputStream stream) throws IOException,
       UnsupportedEncodingException {
         Reader reader = new InputStreamReader(stream, "UTF-8");
         BufferedReader bufferedReader = new BufferedReader(reader);
         return bufferedReader.readLine();
}

/**
 * Encode clientId and clientSecret as base64 basic credentials.
 *
 * Example input: (clientId, clientSecret)
 * Example output: "QWxhZGRpbjpvcGVuIHNlc2FtZQ==".
 */
public final String encodeBasicCredentials(String clientId,
    String clientSecret) {
  return Base64.encodeToString((clientId + ":" + clientSecret).getBytes(),
      Base64.DEFAULT);
}
```

4.2 Mobile Authorization with the Facebook SDK

Facebook provides developers with a SDK for authorizing with Facebook's OAuth implementation. Their implementation does not require WebViews or similar techniques to authorize the application, but rather requires the user to install the Facebook for Android application. We will study how this integration works.

4.2.1 Getting Started

Facebook provides excellent developer guides to getting started with the Facebook SDK for Android [23]. To get started, a developer must first fulfill all the prerequisites, such as having the Android SDK and development tools installed, and the Facebook SDK installed [1]. The developers must then register their Android key hash, which is used to sign the application packages when building your application. Facebook uses this key hash as a security measure to guarantee that the authorization request came from the correct application[2]. This provides a security measure for mobile applications beyond the client credentials. After registering the application in the Facebook Developer Console, you are provided with an App ID and an App Secret. These will be used as client credentials when talking to the Facebook API. All you have to do now is to link the Facebook SDK to you application project, and you can start using the Facebook API.

[1] The Facebook SDK used in these examples is version 3.0.1, available at `https://developers.facebook.com/resources/facebook-android-sdk-3.0.1.zip`.

[2] This is currently only explained in the Application Console, and no linking was possible. The text is: "Your app key hash is required for Facebook Login in order to perform security check before authorizing your app. You can add more than one Key Hashes in case your app is supported on multiple Android platforms."

4.2.2 Simple Authentication with the Facebook SDK and OAuth

In this section we will detail some example code for a simple LoginActivity, in addition to some helpful screenshots. The example will take us through a complete OAuth Authorization Grant flow using the Facebook SDK (See Figure 4.1 for a general overview). This enables the user to authorize third-party applications using native applications on the device for a secure and seamless experience. I will not go into configuration specifics, but the client credentials are stored in `res/values/strings.xml` in plain-text.

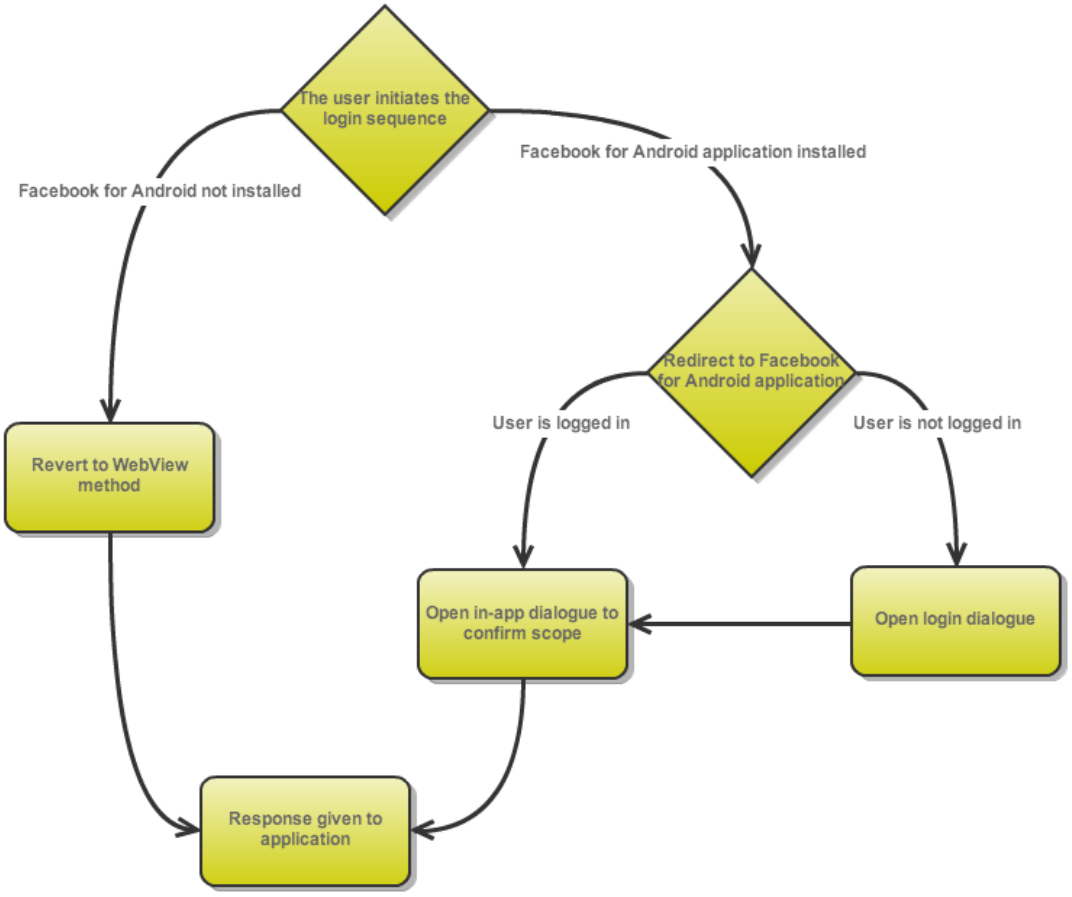

FIGURE 4.1: Facebook SDK General flow

On the next page follows some simple code to create a complete login flow with the Facebook SDK v.3.0.1. We will explain the states this code takes the user in on the next page.

```
public class MainActivity extends Activity {

    TextView textView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        textView = (TextView) findViewById(R.id.welcome);
        // start Facebook Login
        Session.openActiveSession(this, true, new Session.StatusCallback() {

            // callback when session changes state
            @Override
            public void call(Session session, SessionState state,
                    Exception exception) {
                // Only call once Session is fully opened
                if (session.isOpened()) {
                    // make request to the /me API
                    Request.executeMeRequestAsync(session,
                            new Request.GraphUserCallback() {

                                @Override
                                public void onCompleted(GraphUser user,
                                        Response response) {
                                    if (user != null) {
                                        textView.setText("Hello "
                                                + user.getName() + "!");
                                    }
                                }
                            });

                }
            }
        });

    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }

    @Override
    public void onActivityResult(int requestCode, int resultCode, Intent data) {
        super.onActivityResult(requestCode, resultCode, data);
        Session.getActiveSession().onActivityResult(this, requestCode,
                resultCode, data);
    }

}
```

The key to the Facebook SDK is the `Session` class. This class will initiate the authorization sequence, and handle the callback to the application. The authorization sequence is structured much like the browser-based authorization flows in the official specification. The `Session` class fires an `Intent` to the Facebook application, which will redirect the user to the Facebook Application. The SDK behaves differently based on the state of the Facebook for Android application. See Figure 4.2 for a flow chart depicting the different routes.

As we can see from the code example on the previous page, the Facebook SDK provides a fallback to a WebView should the Facebook for Android application not be installed. An example of this can be seen in Figure 4.2. This could potentially leave the implementation open to eavesdropping by the application. If the user has installed the Facebook for Android application, the user is redirected. If the user is not logged in, the user is presented with the Facebook for Android login screen, as shown in Figure 4.3.

After logging in, or if the user already was logged in, the user is redirected back to the application, and presented with a dialogue asking to confirm the permissions (scope) the application is requesting (See Figure 4.3). After the user has made a choice, the user is redirected back to the application through another `Intent` containing an access code, or an error message, should the user have declined the request.

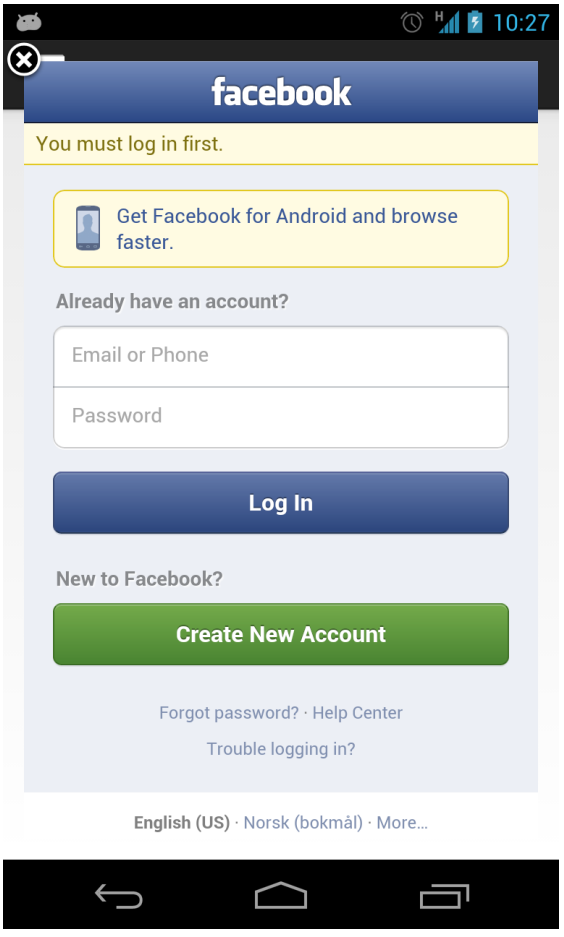

FIGURE 4.2: Facebook SDK WebView flow

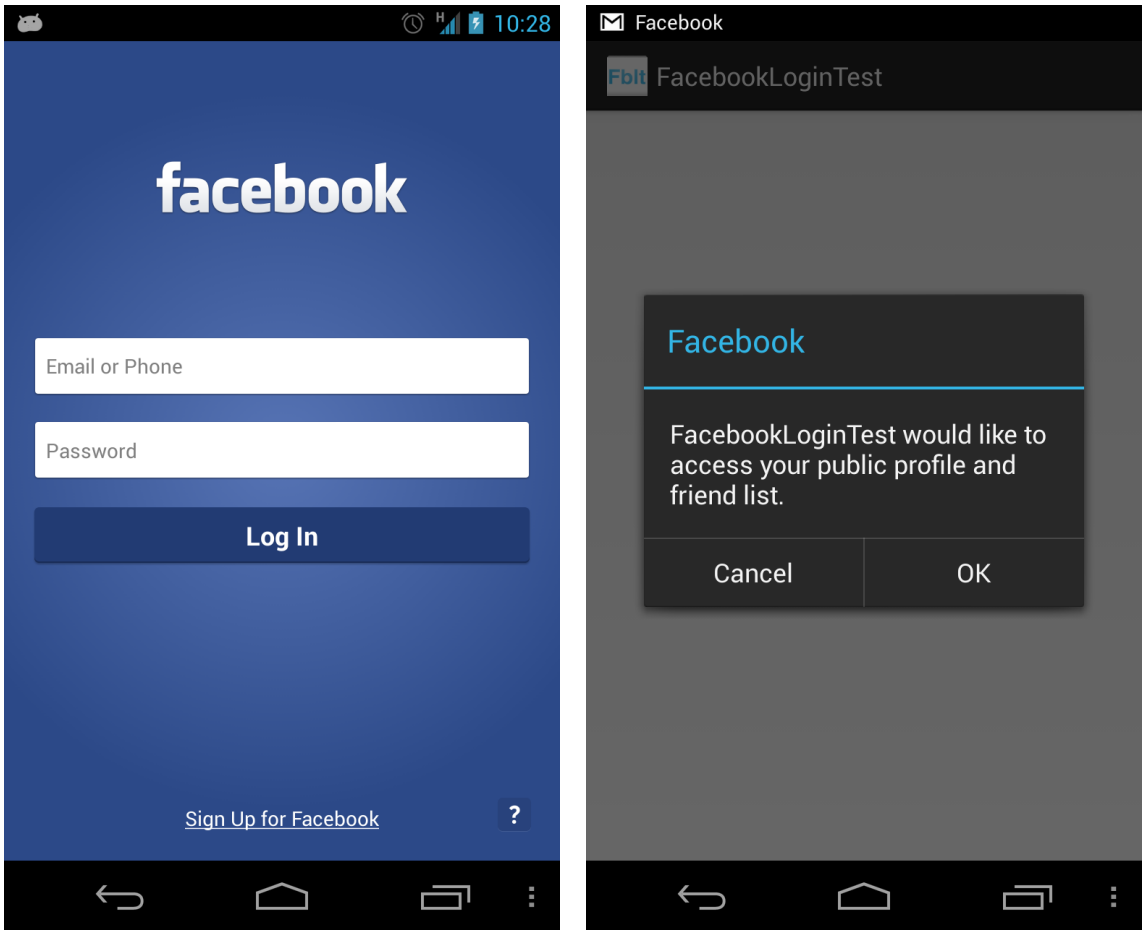

FIGURE 4.3: Facebook SDK native flow

4.2.3 Security considerations and drawbacks

The Facebook SDK implements a redirect flow similar to the browser-based approach, but natively on the Android device. To verify the calling application, the Facebook for Android application verifies that the calling applications digital signature matches the one registered on the Facebook Developer page. Only after verifying the signature is the application allowed to request an access token. To prevent hijacking the Intent, the Facebook SDK will verify that the Intent will be sent to the correct application by verifying the digital signature of the Facebook for Android application. Because the digital signature of an application is unique to the application, and the extracting of said signature is baked into the OS itself, it will be hard to fake this signature for the calling application. It is not clear however, how Facebook verifies that the Facebook for Android application is legit when requesting an access token. However, as the main problem in this case is leaking the user's master credentials, it would in the very least be difficult for an application to fake the Facebook for Android application when the calling application is legit.

While the native approach is secure, the Facebook SDK also supports using the legacy user-agent flow using an embedded browser, as seen in Figure 4.2. This does not verify the application signature, it does not provide any feedback to the user that they are talking to Facebook (and

not some phising site), or even that it is using HTTPS. The application developer can even force the use of this legacy solution, even if the user has Facebook for Android installed. We will look more closely into the drawbacks for such an approach in the next section.

4.3 Mobile Authorization with WebView

The user-agent flow, implemented using the `WebView` class in Android, uses an integrated browser in the application to facilitate the normal redirect-based flow in a browser. The flow most commonly used here is the Implicit flow (See Section 2.2.1.2). This has for a long time been the standard way of performing authorization with OAuth on mobile devices, but the approach has some clear drawbacks, which we will discuss in this section.

4.3.1 Simple Authentication with WebView

A complete authorization flow using WebView requires little code, but requires some fine understanding of the grant flow to obtain an access token. On the next page, example code for how this is done in Android is shown.

```
public class MainActivity extends Activity {

    public static final String CALLBACK_URL = "your callback url";
    public static final String CLIENT_ID = "your client id";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        String url = "https://stagingapi.comoyo.com" + "?client_id="
                + CLIENT_ID + "&response_type=token" + "&redirect_uri="
                + CALLBACK_URL;
        // We can override onPageStarted() in the web client and grab the token
        // out.
        // This will override the callback.
        WebView webview = (WebView) findViewById(R.id.webview);
        webview.getSettings().setJavaScriptEnabled(true);
        webview.setWebViewClient(new WebViewClient() {
            public void onPageStarted(WebView view, String url, Bitmap favicon) {
                String fragment = "access_token=";
                int start = url.indexOf(fragment);
                if (start > -1) {
                    // Access Token acquired.
                    String accessToken = url.substring(
                            start + fragment.length(), url.length());
                }
            }
        });
        webview.loadUrl(url);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }

}
```

Key to this implementation is the `WebView` and `WebViewClient` classes. These implement a browser inside the application itself, which will direct the user to the token endpoint for authentication and authorization of our application. However, to extract the token, the developer must intercept the response from the OAuth endpoint and extract the access token from the URL.

4.3.2 Drawbacks

The main problem with this method is that to extract the token, the developer must intercept the network traffic. However, `WebViews` that enable Javascript can also inject arbitrary Javascript into the `WebView` itself, making it possible to listen to the user's input. A malicious developer might add Javascript to listen to the `<input>` element for key events, and send those off to some remote server. Authorization through `WebViews` on Android is therefore inherently insecure, and should be regarded as malicious by default. An example of how this is done is shown below.

```
@Override
public void onPageFinished(WebView view, String url) {
    view.loadUrl("javascript:(function() { " +
            "document.getElementsByTagName('body')[0].style.color = 'red'; " +
            "})()");
}
```

The above code will make all the text on the page red, which shows that arbitrary javascript can easily be injected when the page is finished loading, without the user receiving any feedback. This is also true for the FacebookSDK using WebView. A developer can easily edit the code of the FacebookSDK library and inject arbitrary Javascript.

4.4 Native Mobile Authorization Grant

To create a native flow for mobile clients, many companies, including Comoyo, use the Resource Owner Password Credentials Grant (See Section 2.2.1.3). I will call this the native mobile authorization grant. The native mobile authorization grant is best fit for applications that a company controls, or has a contractual agreement towards, and is not suited for third-party developers, as it defeats the purpose of not letting the application know the password. However, as we will demonstrate later, it can be used by an AccountManagerService to create an even better authorization flow.

The flow is similar to a regular HTTP Basic Authentication scheme where the username and password is sent to an API endpoint as parameters, and a cookie containing an opaque string is set on the browser session and included on any subsequent request to the server. This solution will enable both authentication and authorization of an end-user using the same OAuth scheme as any other application would need to use to enable access to the API, but without using integrated browsers in the application. Figure 4.4 shows a flow chart depicting how the application will obtain an access token from the resource owner credentials.

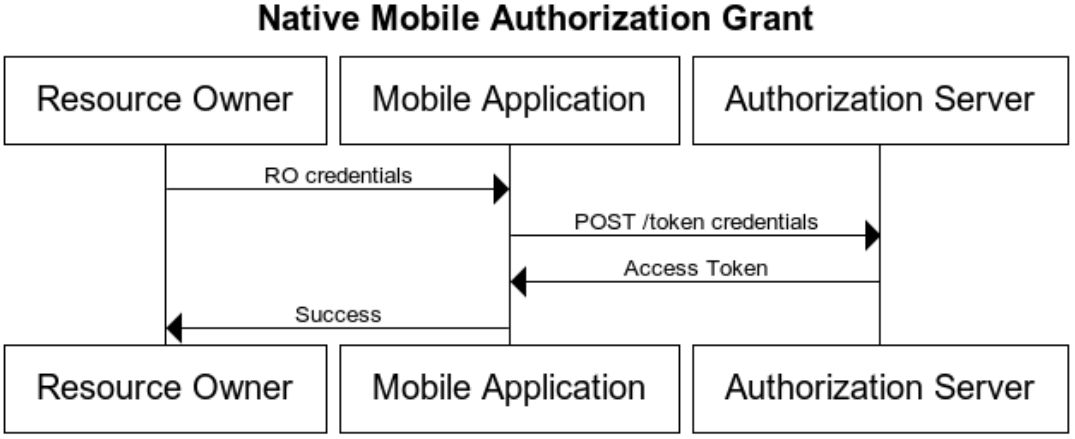

Figure 4.4: OAuth 2.0 Proposed Native Mobile Authorization Grant flow

In short, the flow exchanges the resource owners credentials for a time- and scope-limited access token that can optionally be refreshed using a refresh token. This enables a resource owner, or

the API provider to revoke access on an application specific basis in case of misuse. However, this solution comes with a big trade-off, since the user must trust the application developer with his master credentials, something which OAuth was designed to prevent to begin with. However, this can be mitigated using application-specific credentials, much like Google uses for unsupported applications in its two-factor authentication scheme [24]. These are generated, one-time use only, and must be manually inputted into the application. These credentials can be much the same as access tokens, but should be more readable, and do not require an as high entropy as access tokens[3]. This approach can be compared to a redirect similar to the authorization code grant (Section 2.2.1.1), where the user-agent used for redirection is the resource owner herself, and the one-time credentials can be seen as the authorization code.

This flow is similar to Twitter's xAuth flow for OAuth 1.0 enabled applications wanting to access their API [25].

4.4.1 Example Implementation

The following shows how to obtain an access token using the client native mobile authorization grant using cURL. Line endings added for sake of presentation.

```
curl https://stagingapi.comoyo.com/oauth/token
--user <client_id>:<client_secret>
--data "grant_type=password&username=<username>&password=<password>"
-H "Accept: application/json, text/plain, */*"
```

The sample HTTP requests looks like the following, where the SSL handshake has been omitted for display purposes:

```
> POST /oauth/token HTTP/1.1
> Authorization: Basic <Base64 encoded credentials>
> Host: stagingapi.comoyo.com
> Accept: */*
> Content-Length: 66
> Content-Type: application/x-www-form-urlencoded

grant_type=password&username=<username>&password=<password>
```

This can be translated into Java to work in our test environment on an actual device as shown on the next page.

[3]The current recommended entropy for access tokens is 128 bits or more.

```
private class OAuthAuthenticator extends AsyncTask<String, String, String> {

  @Override
    protected String doInBackground(String... params) {
      String username = params[0];
      String password = params[1];
      return auth(username, password);
    }

  private String auth(String username, String password) {
    URL url = null;
    InputStream inputStream = null;
    String response = null;
    try {
      String base64credentials = encodeBasicCredentials(
          getApplicationContext().getResources().getString(R.string.client_id),
          getApplicationContext().getResources().getString(R.string.client_secret));
      url = new URL(getApplicationContext().getResources().getString(
            R.string.oauth_url));
      HttpsURLConnection conn = (HttpsURLConnection) url.openConnection();
      conn.setReadTimeout(10000 /* milliseconds */);
      conn.setConnectTimeout(15000 /* milliseconds */);
      conn.setRequestMethod("POST");
      conn.addRequestProperty("Accept", "application/json");
      conn.addRequestProperty("Authorization", "Basic " + base64credentials);
      conn.setDoInput(true);
      conn.setDoOutput(true);
      List<NameValuePair> nameValuePairs = new ArrayList<NameValuePair>(3);
      nameValuePairs.add(new BasicNameValuePair("grant_type", "password"));
      nameValuePairs.add(new BasicNameValuePair("username", username));
      nameValuePairs.add(new BasicNameValuePair("password", password));
      UrlEncodedFormEntity params = new UrlEncodedFormEntity(nameValuePairs);
      params.writeTo(conn.getOutputStream());
      conn.connect();
      if (conn.getResponseCode() / 100 % 2 != 0) {
        Response was unsuccessful
          return null;
      }
      inputStream = conn.getInputStream();
      response = stringify(inputStream).trim();
    } catch (MalformedURLException e) {
      e.printStackTrace();
    } catch (NotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
    return response;
  }
```

Note that the above code does not validate the SSL certificate of the server. This was left out to make the code example smaller, but leaves the implementation open to man-in-the-middle attacks.

4.4.2 Advantages

This flow allows for a browserless authorization flow for mobile devices. The flow itself is a simple HTTP Post call, which is simple to implement on mobile devices. It is easily testable and understandable. This flow has advantages over simple HTTP Basic Authorization schemes since

it allows you to use the existing OAuth backend infrastructure, which is required to do proper and secure API authorization for browser-based applications.

4.4.3 Drawbacks

In this section we will review the identified drawbacks of the proposed solution, and suggest potential mitigations for these.

4.4.3.1 Mobile Devices are Public Clients

As stated in the OAuth specification, mobile devices should be regarded as public clients, because they are installed on a client device, and therefore cannot keep client credentials secret [7]. Through decompilation of the application package using tools such as android-apktool [4], a developer can decompile, rebuild code to near original order, and extract client credentials from the application itself. This was recently done to extract client credentials from Twitter's Android application, making it possible for third-party developers to fake their application as Twitter's own, and therefore circumventing rate-limiting imposed on third-party developers [13].

There is currently no way to securely mitigate this risk. The only possibility is security by obscurity, by obfuscating the secret string, and making it costly to reverse-engineer it.

In short, the server cannot trust that the request for an access token comes from a pre-approved mobile client, and we must assume that any special permissions given to pre-approved clients will be available to all clients.

4.4.3.2 Scope must be pre-approved

In this proposed flow, the scope must be pre-approved, as it is pointless to let the application be responsible for showing the requested scope to the user, and obtaining permission. Since this easily can be circumvented, the client's scope must be pre-approved by the API-provider, and be as small as possible. This follows the principle of least privilege, which is a basic security principle any application should follow.

To protect the user from mischievous use, especially when considering that mobile devices are public clients, and the consequences of that, the API provider must also carefully consider which endpoints and data that will be exposed to the application. Endpoints providing payment or for privacy reasons, such as containing medical records, or highly person sensitive information, should be kept safe, and should therefore not be exposed to mobile applications.

4.4.3.3 Resource Owner must input credentials to application

The original purpose of OAuth was to prevent resource owners from having to share their master credentials with a third-party application to facilitate API access. However, with this proposed

[4] Android APKTool is available at `https://code.google.com/p/android-apktool/`

authorization flow, the resource owners credentials has to be shared with the application. The difference is that the resource owners credentials should not be stored on the device itself and will instead be replaced by a time- and scope-limited access token that also need the client's credentials in order to be used.

Because the resource owner password credentials flow exposes the user's credentials to the client, it must only be trusted clients that are allowed to use this flow. While this in theory poses a risk to some users, the flow does not increase or decrease the chance of phishing attacks. Uninformed users will most likely still input their master credentials in applications where they should not. Keyloggers would be able to sniff the password regardless of authorization flow in OAuth 2.0.

4.5 Mobile Authorization with the Google SDK

Google's solution to OAuth for mobile applications is strikingly similar to Facebook's, but provides deeper OS-level integration on the Android platform. In addition, Google's solution gives the user a completely different context when approving the scope of the request, which is different from Facebook that retains the approval process in the calling application. In this section, we will examine the Google Play Services in detail.

4.5.1 Getting Started

Similar to Facebook, Google requires that an application developer uploads the signing certificate fingerprint (SHA1) for the signing key, so that it can be validated in the authorization process. The SHA1 fingerprint can be obtained by using the keytool utility, as shown below.

```
keytool -exportcert -alias androiddebugkey -keystore <path> -list -v
```

The fingerprint is then provided to Google through the developer console on Google's web pages (See Figure 4.5). Google provides an SDK, much like Facebook, which needs to be included in the application. The exact process to integrate the Google Play Services in the application itself is subject to change. Google does not specify which version of the SDK it uses, but it requires Google+ to be installed on the client device to function.

Unlike the Facebook SDK, Google does not require that client credentials are present in the application, instead relying on the fingerprint hash and package names to be reported correctly from the `PackageManager`. Below you will see the necessary code required to integrate with Google Play Services, before we demo a working Sign-In example.

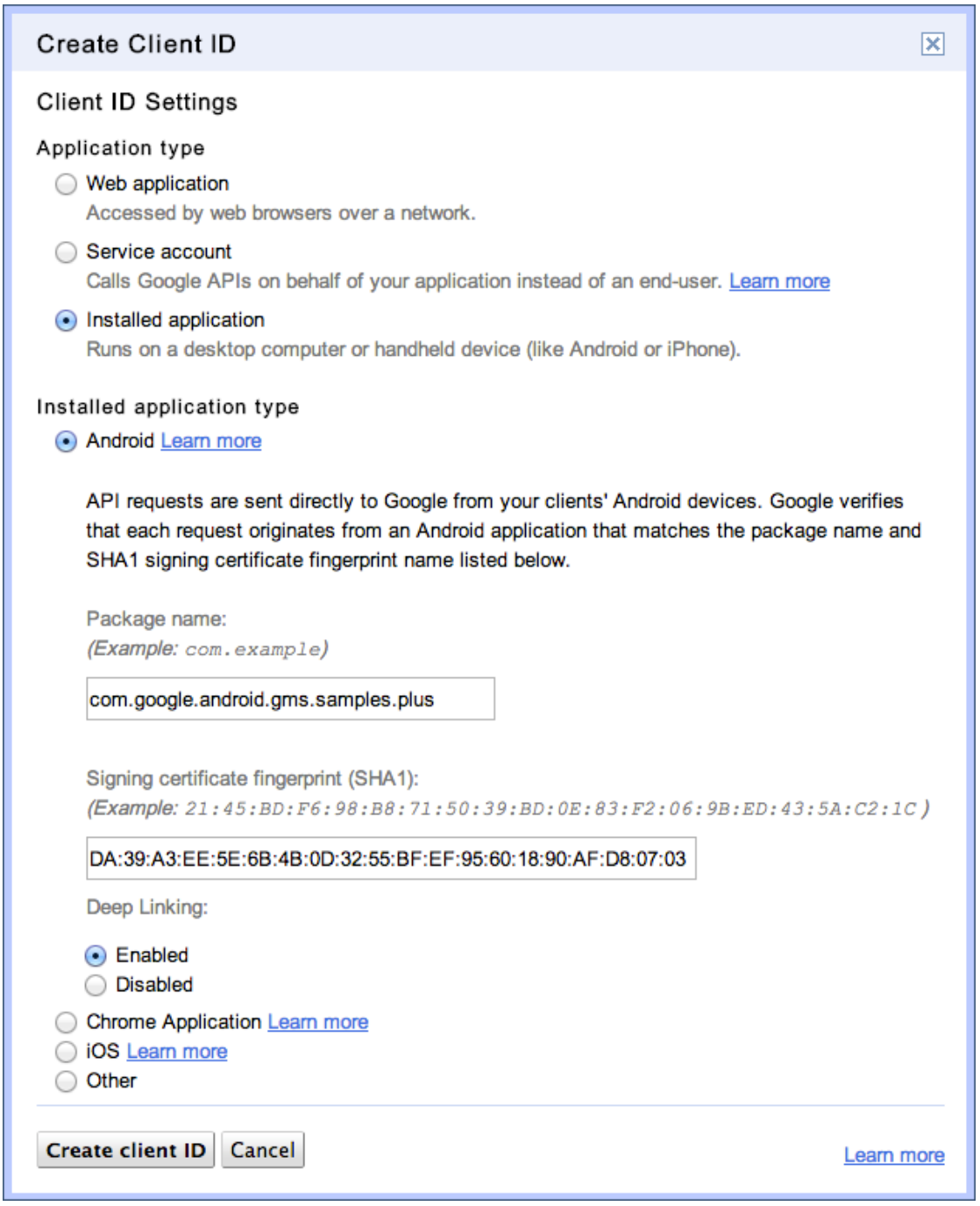

FIGURE 4.5: Google Developer Console - configure new application

```
import com.google.android.gms.common.*;
import com.google.android.gms.common.GooglePlayServicesClient.*;
import com.google.android.gms.plus.PlusClient;

public class GoogleServiceIntegrationActivity extends Activity implements
        View.OnClickListener, ConnectionCallbacks, OnConnectionFailedListener {
    private static final String TAG = "GoogleServiceIntegrationActivity";
    private static final int REQUEST_CODE_RESOLVE_ERR = 9000;

    private ProgressDialog mConnectionProgressDialog;
    private PlusClient mPlusClient;
    private ConnectionResult mConnectionResult;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mPlusClient = new PlusClient.Builder(this, this, this)
                .setVisibleActivities("http://schemas.google.com/AddActivity",
                        "http://schemas.google.com/BuyActivity").build();
        // Progress bar to be displayed if the connection failure is not
        // resolved.
        mConnectionProgressDialog = new ProgressDialog(this);
        mConnectionProgressDialog.setMessage("Signing in...");
    }

    @Override
    protected void onStart() {
        super.onStart();
        mPlusClient.connect();
    }

    @Override
    protected void onStop() {
        super.onStop();
        mPlusClient.disconnect();
    }

    @Override
    public void onConnectionFailed(ConnectionResult result) {
        if (result.hasResolution()) {
            try {
                result.startResolutionForResult(this, REQUEST_CODE_RESOLVE_ERR);
            } catch (SendIntentException e) {
                mPlusClient.connect();
            }
        }
        // Save the result and resolve the connection failure upon a user click.
        mConnectionResult = result;
    }

    @Override
    protected void onActivityResult(int requestCode, int responseCode,
            Intent intent) {
        if (requestCode == REQUEST_CODE_RESOLVE_ERR
                && responseCode == RESULT_OK) {
            mConnectionResult = null;
            mPlusClient.connect();
        }
    }

    @Override
    public void onConnected() {
        String accountName = mPlusClient.getAccountName();
        Toast.makeText(this, accountName + " is connected.", Toast.LENGTH_LONG)
                .show();
```

```
    }

    @Override
    public void onDisconnected() {
        Log.d(TAG, "disconnected");
    }
}
```

4.5.2 Simple Authentication with the Google Play Services SDK and OAuth

Using the code example in Section 4.5.1, we can extend it to include a simple Sign-in flow by adding the code below.

```
// In your activity's layout specification
<com.google.android.gms.common.SignInButton
    android:id="@+id/sign_in_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

// Register an OnClickListener in your Activity
findViewById(R.id.sign_in_button).setOnClickListener(this);

// In your activity
@Override
public void onClick(View view) {
    if (view.getId() == R.id.sign_in_button && !mPlusClient.isConnected()) {
        if (mConnectionResult == null) {
            mConnectionProgressDialog.show();
        } else {
            try {
                mConnectionResult.startResolutionForResult(this, REQUEST_CODE_RESOLVE_ERR);
            } catch (SendIntentException e) {
                // Try connecting again.
                mConnectionResult = null;
                mPlusClient.connect();
            }
        }
    }
}
```

Google's authorization flow is similar to Facebook's, and doesn't prompt you for a password if you are already logged in to your device. In Figure 4.6 and 4.7 you can see the typical authorization flow. Note that Google supports multiple accounts, as shown in Figure 4.6. Google does not enforce the user to leave the context of the application, but uses dialogues, much the same way Facebook does. Google's implementation uses the built in `AccountManager`, which is a native way of implementing Accounts in android. `AccountManager` also supports issuing Tokens, and any implementing service can override this method to provide tokens to applications through a standardised API for Android developers.

4.5.3 Security considerations and drawbacks

The security considerations and drawbacks of the Google Play Services Authorization flow are the same as the Facebook SDK, but with one notable exception. Google does not fall back to

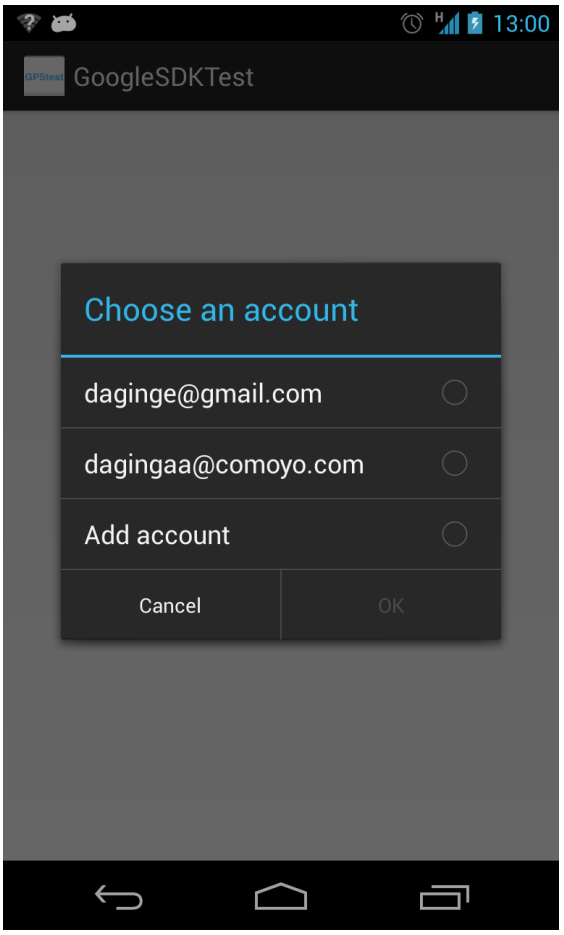

FIGURE 4.6: Facebook SDK WebView flow

the insecure WebView flow, meaning that from a user's perspective, everything will look native. It is also harder for the calling application to eavesdrop on the credentials, since the credentials are stored in another application, protected by the Android Security model. Google's approach might also be better in terms of training user's not to trust WebViews, as it will not "look" native, and therefore breaks their preconception of what is going to happen, hopefully raising questions.

4.6 Summary

While academia could provide little or no relevant literature, the industry itself seem to have solved many of the problems with mobile authorization. Both Facebook and Google have recently launched new SDKs which greatly simplify the authorization process, while simultaneously pushing their own applications (Facebook for Android or Google+ is required). These solutions use standard features of the Android platform, such as `AccountManagaer`, which provides a common interface for Account Management for the user. The user's credentials are stored using current industry best practices for credentials storage on mobile devices.

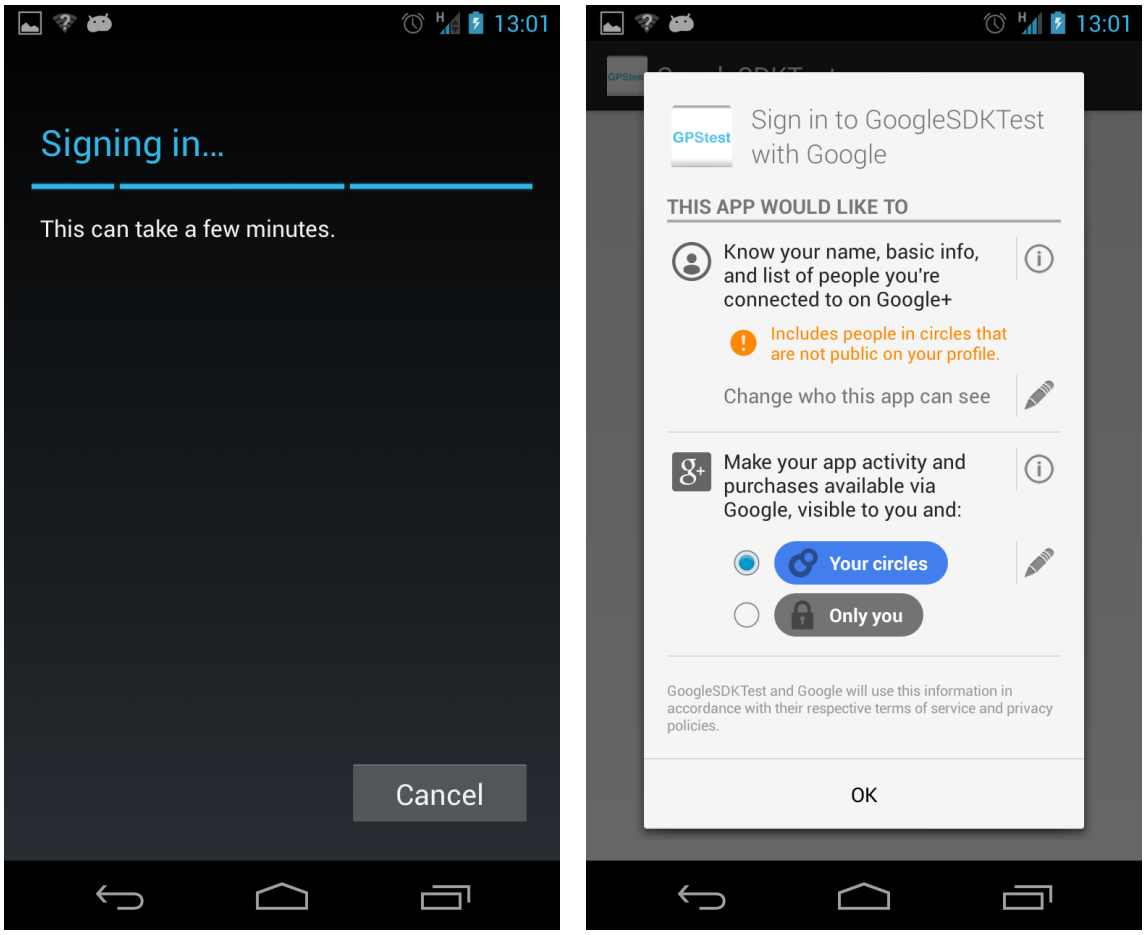

FIGURE 4.7: Google Play Services Sign in and Authorization flow

The main problem right now is securing the authorization application, and the communication between client and server. By combining the Native Mobile Authorization Flow, detailed in Section 4.4, we can at least bypass the WebView flow. While the communication between client and server can still be spoofed by an attacker, at least the user's credentials are safe from malicious applications, as is the intention of OAuth. However, as Facebook and Google's implementation are closed source, I need to implement an example client authorization agent to evaluate the security practices of such an approach.

Chapter 5

Native Mobile Authorization Client

5.1 Introduction

This section will detail the implementation of a mobile client that can facilitate authorization requests using OAuth on the Android platform. As shown in Chapter 4, and concluded in Section 4.6, to examine the security and viability of such an approach, I need to implement a solution.

The solution will use the built-in `AccountManager` in Android, which will register our application as a new Account type in Android, enabling other applications to use the built-in API to interface with my solution. This will mainly be a proof-of-concept, detailing some of the steps one can take to secure the communication between client and server, and simultaneously secure the user's credentials. Some implementation details may be omitted for brevity.

5.2 Prerequisites

The implementation will follow the same rule as the Test environment, as detailed in Section 1.6. To help create the application, the guide to custom account types in Android [26] has been followed.

To simplify the application code, I will not include cryptographic solutions to store the user's credentials in the AccountManager, but will instead discuss different security strategies to secure the application in Section 5.5.

5.3 Terminology

This is a short list of the terminologies used in this chapter.

Client sometimes referred to as the calling application is the application that wishes to interface with OAuth.

Authorization Application (AA) is the entity that handles scope granting, user authentication, and communication with the Authorization Server on the mobile device, and provides a common API for clients to interface with.

Authorization Server (AS) is the same as the Authorization Server as defined in Section A.1

AccountManager The common interface provided by Android to talk to Services providing Authentication and Authorization services on the device.

5.4 Native Application Flow

The main idea behind the native application flow is a variation of the Authorization Code Grant Flow. An Android application is not particularly good at keeping state between calls using `Intents`, so we need to rely on a "user-agent", or in the case, the calling application, to relay messages between the authorization service and the authentication service. Fortunately, this can easily be done by the calling application, and is the recommended solution for OAuth authorization put forth by Google. The abstract flow can be seen in Figure 5.1.

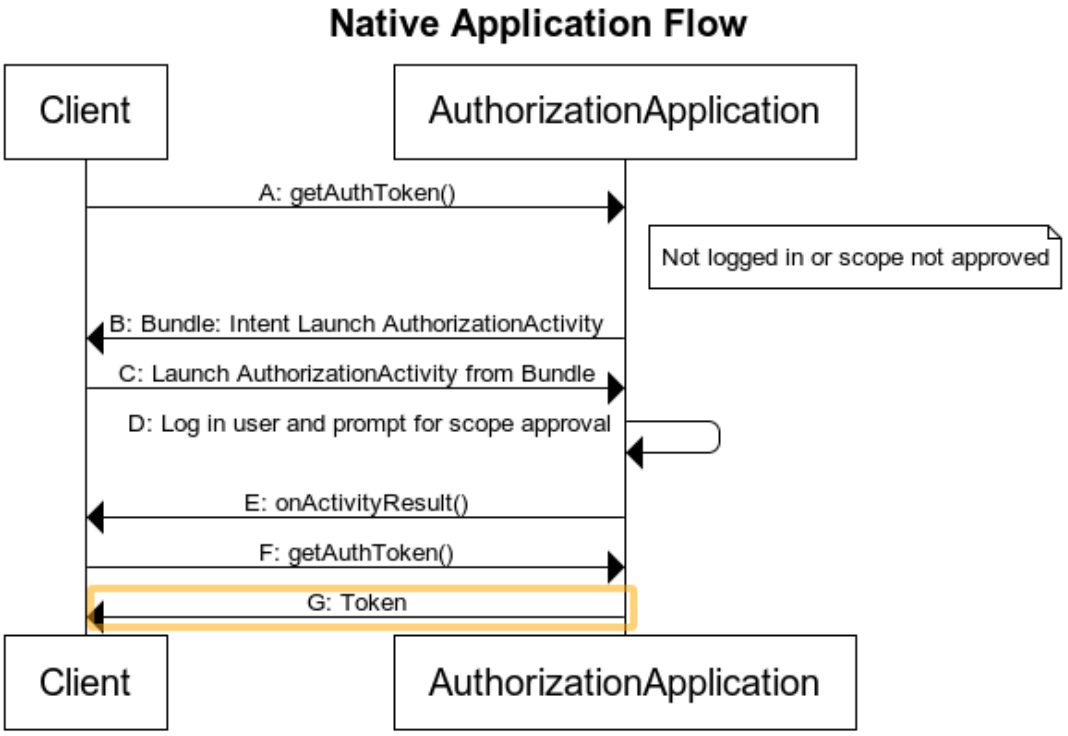

FIGURE 5.1: Abstract Native Application Flow

A. The client application calls getAuthToken on the `AccountManager`, which will call our application.

B. If the user is not logged in and/or the scope is not approved, the authorization application should return a `Bundle` with an `Intent` for the `AuthorizationActivity`.

C. The client should `startActivityForResult()` with the result from the `Bundle`, launching the authorization activity in the authorization application.

D. The authorization activity must authenticate the user and prompt for scope approval. This requires communication with the server, and the server should issue a one-time authorization code which can be redeemed for an access token. Additional security measures are needed for storage of this authorization code.

E. The authorization application calls the `onActivityResult()` in the client.

F. The client calls the `getAuthToken` method again to obtain the access token. It is important that only a verified app can redeem the access token.

G. The token is returned to the client.

The main idea is that the calling application will relay the `Intent` from the authorization application, much like the browser-based flows. This will make the client application the user-agent for the resource owner, without disclosing any credentials or tokens ahead of time.

5.4.1 Setting up the calling application to relay Intents

To enable the calling application to relay the `Intent`s returned from the `getAuthToken()` call we need the following code.

```
private class OnTokenAcquired implements AccountManagerCallback<Bundle> {
  @Override
  public void run(AccountManagerFuture<Bundle> result) {
    ...
    Intent launch = (Intent) result.getResult().get(AccountManager.KEY_INTENT);
    if (launch != null) {
      startActivityForResult(launch, 0);
      return;
    }
  }
}
```

The code uses `startActivityForResult` to enable the client's activity to capture the result of the `Intent`, either an error, or a successful reply, by implementing `onActivityResult`. From this method, in the case of a successful reply, the client can call the `getAuthToken` method again to obtain a valid access token.

5.4.2 Authorization Application Architecture

To enable the authorization flow using an `AccountManager` type flow, the authorization application must implement a few interfaces and create a new `Service` that can be registered with the operating system. In Figure 5.2 we see what classes and methods must be implemented.

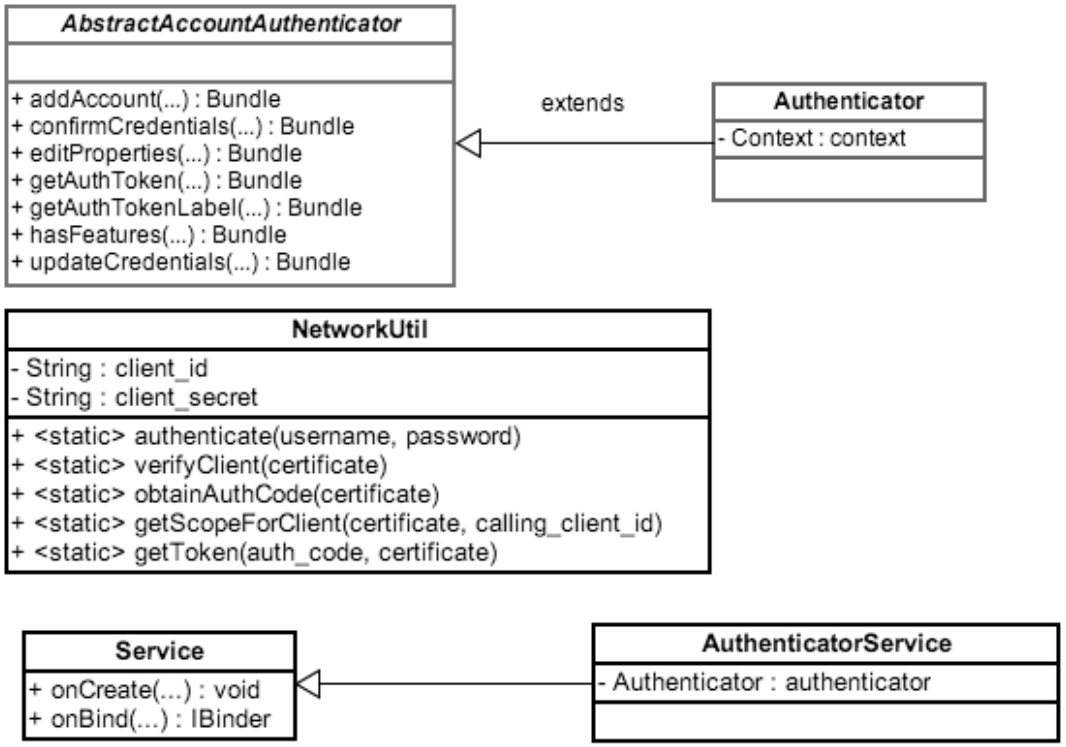

FIGURE 5.2: Class Diagram for Essential Classes in the Authorization Application

The `Authenticator` class extends the `AbstractAccountAuthenticator` to present a common interface for the `AccountManager` to interact with. The methods implemented by this class is called when the client calls e.g. the `getAuthToken()` method.

The `NetworkUtil` class has a set of static methods that will interact with the server. These should be called by the `Authenticator` and `AuthenticatorActivity` (not shown) when they want to interact with the server.

The `AuthenticatorService` extends the `Service` abstract class, which will tell the operating system that this application provides a Service. The implementation is short, and just provides the `IBinder` instance of the `Authenticator` class in the `onBind()` method which is called by the operating system to register the service. This, combined with the `AbstractAccountAuthenticator`, will tell the operating system that an authenticator service is provided, and add an account type to the `AccountManager` interface. See the next page for an implementation example.

```
public class AuthenticatorService extends Service {

  private Authenticator authenticator;

  @Override
    public void onCreate() {
      this.authenticator = new Authenticator(this);
    }

  @Override
    public IBinder onBind(Intent intent) {
      return authenticator.getIBinder();
    }

}
```

5.5 Securing the solution

5.5.1 Storing the user's credentials

The AccountManager is not a keychain or secure credentials store, and the naive implementation will store the user's credentials in plain text, as the sample application does. This is in most cases a bad idea, as it enables other applications to fetch the user's credentials. To mitigate this threat, one can implement a simple cryptographic solution using symmetric keys, prompting the user for a password to decrypt the credentials before use. To increase the security, the client could also exchange the user's credentials with an access token, which would get passed to authenticate the user, and only allow certain scopes on mobile devices, to limit the potential for misuse.

5.5.2 Securing communication channel between AA and AS

The most critical issue with this solution is securing the communication between the Authorization Application (AA) and the Authorization Server (AS). The AA must be able to verify that it is talking to the true AS, to avoid man-in-the-middle (MITM) attacks. This can be done by verifying the SSL certificate of the server in the client side code. By using a trusted Certificate Authority, the client can ask this third party to verify the validity of the issued certificate, thereby guaranteeing that the client is talking to a legit server. Since the communication channel is secured with TLS 1.0 or greater, it is not possible to eavesdrop on the communication by a third-party, e.g. a malicious application installed on the phone.

However, this is also true for the server. Since the application we are developing requires special permissions, such as handling the scope granting on the client device, the server must ensure that it can trust the identity of the application talking to it as well. In traditional OAuth, this is done by issuing a client_id and client_secret to the client, but in the case of a native application running on a client device, the client is to be regarded as a public client, i.e. a client not capable of protecting its secret. This raises issues with the current solution. There is currently no way

of securely storing a secret in the Android Application Package [3], and the only way to deter attackers from finding the secret is by obfuscation techniques, which will only make it harder for an attacker, not impossible [27].

However, there is a way of creating an additional layer of security. By using a client-side SSL certificate, which the server can validate, in addition to the client_id and client_secret with obfuscation techniques, we can make it very hard for an attacker to replicate the communication between our application and the server [3]. If we in addition to this use the public certificate as the client identifier, we can make it almost impossible to decipher the APK after compilation to fetch the necessary information. Even if the APK is rebuilt with malicious code, it would not be able to talk to the server, as the signing certificate would have changed.

But, as with all security measures, taking preemptive measures, in this case by not allowing certain sensitive scopes to be available to public clients in the first place, is the best approach.

5.5.3 Verifying the client's signature

A big part of the puzzle to a secure solution is to verify that the calling application (client) is registered as a valid OAuth client, and can obtain access tokens. We want to avoid that malicious applications can obtain a "secret", and pose as the real application. We do this by requiring that the client developer has registered his public certificate with the Authorization Server. As mentioned in Section 2.3.3, each Android application is uniquely signed by a key generated by the application developer. Only if an attacker is able to obtain the keystore containing the signing secret and the keyphrase to unlock the keystore, which is stored on the developer's machine, is the attacker able to dupe this information.

The only way an attacker can circumvent this protection is if the client device has been rooted, and the `PackageManager` service can no longer be trusted. Since this is a cumbersome and manual process, requiring both physical access to the device, and a wipe of the client device, we can assume that a third-party attacker is not able to circumvent this protection [3].

To get the calling applications unique signature, we can use an approach made possible by the `AccountManager` interface and the `Service`-based approach to calling `getAuthToken()`. As mentioned in Section 2.3.2, each Android application is assigned a unique identifier at install time, the UID. By using the code below, we can get the UID, and ask the `PackageManager` for the public certificate of this application. Since the `Binder` approach[1] is used, we can extract the UID of the calling application using `Binder.getCallingUid()`. The code on the next page shows how we can extract the certificate with this information.

[1]Using Binder or Messenger is the preferred mechanism for RPC-style IPC in Android. They provide a well-defined interface that enables mutual authentication of the endpoints, if required.

```
/**
 * Extract the signature of the application with a given UID as a {@link String}.
 * @param uid The UID of the application
 * @return An ASCII representation of the signature for the application
 */
private String extractCertificateFromUid(int uid) {
  try {
    String packageName = context.getPackageManager().getPackagesForUid(
        uid)[0];
    PackageInfo packageInfo = context.getPackageManager()
      .getPackageInfo(packageName, PackageManager.GET_SIGNATURES);
    return packageInfo.signatures[0].toCharsString();
  } catch (NameNotFoundException e) {
    return null;
  }
}
```

Chapter 6

Discussion & Conclusion

6.1 Discussion

I have defined five research questions that need be answered by this thesis. I will discuss my findings to each of these research questions in this section.

6.1.1 Research Question 1

What methods are used today for authentication and authorization on mobile devices leveraging OAuth 2.0?

In this thesis, I have examined four approaches to mobile authorization flows using OAuth 2.0. These are, in no particular order, WebView, Resource Owner Authorization Grant, Facebook and Google. While the four methods vary greatly in implementation complexity and approach, they all reach the same goal, obtaining an access token to act on behalf of the user without the user's master credentials.

The four methods represent the most common scenarios for how a developer would implement OAuth on a native device. While Facebook and Google's methods are similar, Facebook supports a fallback to a more insecure approach when their authorization application is not present on the device. A developer can also force the use of this approach, should she choose to do so.

I did not find any relevant literature or hard data on the real-world usage of either approach, but as Google and Facebook, the world's biggest API providers for third-party developers, now use native implementations in their SDK, we can assume that these methods are used the most. However, the `WebView` approach of integrating a browser in the application itself has been the most common approach, and is the easiest for client developers to start using, as it requires no external SDK to use.

There are three approaches to mobile authorization using OAuth, the WebView approach, the Resource Owner Password Credentials flow, and the Native Mobile Authorization Client flow.

6.1.2 Research Question 2

Are there known attacks or vulnerabilities in today's solutions, and how are they executed?

Of the four methods examined, I found several known vulnerabilities or "gotchas" that need to be addressed. The most insecure of the four methods was by far the `WebView` approach, as Android allows arbitrary JavaScript injection in `WebView`s which can sniff out credentials from the user. This method is also supported, and can be forced, when using the Facebook SDK. No such vulnerability was found in the Google SDK.

While the Resource Owner Credentials Grant approached used in the Native Mobile Authorization Flow provides a simple approach to obtaining an access token for an application, it does not ensure that the user's credentials are protected from the application, and should as such only be used by developers of which a company has a contractual agreement with. The scope must be pre-approved, and should follow the principle of least privilege. As stated in the specification, this approach is great for migration purposes, but defeats the purpose of OAuth which is to protect the resource owners master credentials.

The biggest threat to OAuth security is the WebView approach, which enables a client developer to inject arbitrary JavaScript into the WebView to snoop for credentials. There are also numerous security "gotchas" related to storing the user's credentials or tokens on the client device, especially if the device is "rooted". Securing communication between authorization client and server has also proven to be difficult, or even impossible using today's solutions.

6.1.3 Research Question 3

How can we, or can we, implement an authorization flow without using browser-specific solutions, such as WebView on Android?

Of the four implementations studied, three of them supported a native flow for obtaining an access token. Contrary to popular belief, OAuth does support a native grant flow, but requires that the user enter his credentials in the application itself, or that the user installs an application which provides an authorization service. The communication between client and server still requires the user to input his credentials on the device itself. This is still more secure than using the `WebView` method of authorization, should the application be malicious. While neither Facebook or Google removed the user from the context of the application for the scope granting, the user was sent to the respective application for credentials entry should it be necessary.

Using the native mobile authorization client approach, it is possible to implement an authorization flow on mobile devices without using WebViews. Both Facebook and Google has implemented such an approach

6.1.4 Research Question 4

How can we, or can we, securely store client_id and client_secret in applications distributed through Google Play (Android Market)?

There is no way to securely store a secret key in an application on a mobile device running Android, at least not in a way that is guaranteed to be impossible to break. While there are many good approaches to cryptographically storing information on the device post-install time, such as using the Bouncy-Castle Keystore with a secret key distributed outside the application context, or set by a user, there is no way to include a secret key in the application package (.apk) distributed through the Android Market or elsewhere. While one can resort to obfuscation techniques that make it harder for an attacker to extract such credentials, they are by no means impossible to extract, given enough time and effort [3] [27].

It is worth noting that the main reason why one would want to do this in the context of OAuth is to secure the communication between client and server, which could uniquely identify the client application to the server. While I have discussed measures such as client-side SSL certificates and additional measures such as obfuscation and signed application packages (See Section 5.5), none of these ways, even combined, guarantee that the communication can only come from known clients.

There is no way to securely store client credentials or "secrets" in application distributed through Google Play. For the sake of OAuth, we have other ways of identifying a calling application, through the use of signature certificates and package names, but these methods are not bullet proof. The best approach is limiting the amount of information available to a mobile application using the principle of least privilege.

6.1.5 Research Question 5

How can we improve the solutions to authorization on mobile devices that exists today?

Of the four solutions examined, both Facebook and Google's approach to authorization seem secure enough for consumer use. However, Facebook allowed developers to rely on a `WebView`-only approach, which is insecure.

I implemented a proposed solution containing several best practices and demonstrated them in Chapter 5, where I also demonstrated how to follow some best practices with regards to credentials storage. I believe Google's approach, launched in November 2011, accurately show the current best practice of OAuth on mobile devices.

A significant drawback of these methods are that they require the user to install an additional application on their device. For Google, which controls Android on most consumer devices, this is not a problem, as they can bake their applications into the OS itself, but for other parties, this requires an effort from the user, possibly making the usage of such API's less desirable on mobile devices for third-party developers. These best-practices also require developer effort from the API providers standpoint, as an SDK and authorization application must be developed and maintained, and the security aspects of such an application are hard to get right. Developers of such applications should opt for a security audit by a seasoned professional before launched the solution to the public.

By removing the possibility of falling back to a WebView-approach, we can make the solution more secure and robust to attack. By following current best practices on Android for credentials storage, use obfuscation and client-side SSL certificates when communicating with the server, we can make the solution more secure. However, there is no 100 percent secure method of authorization using OAuth on mobile devices.

6.2 Conclusion

While the topic of security on mobile devices, and mobile-friendly authorization flows for OAuth rages on in blog posts [10] and in conferences [11], it seems that the issue is a lot less serious than some advocates would have us believe. While academia has performed few studies on using authorization frameworks on mobile devices as of March 2013, it seems that the industry already has fairly secure approaches implemented. Both Facebook [23] and Google [28] offer secure, native approaches to mobile authorization for third parties. These approaches are fairly recent, with Google releasing their approach as late as November 2012.

However, these solutions do demand more resources from developers of said companies as they must maintain SDKs and applications on the Android market to facilitate such authorization flows, when compared to the more mainstream approach of using `WebView`'s. Nonetheless, the usage of `WebView`'s is an anti-pattern, and a security risk for the user, as the user will be accustomed to entering their credentials in third-party applications, where no additional security checks can be made to prevent credentials leakage to a malicious application.

A developer seeking to implement OAuth on mobile devices must first carefully consider the security consequences of allowing public clients access to protected APIs. One must always assume that any application can mimic another application given enough time and resources. Thus, applications with "special" privileges must be avoided at all costs. An example of such an application is the Twitter client which can bypass rate-limiting to Twitter's API. After these consequences have been considered, the developer must decide on the resources that should be spent, and the purpose of having native authorization flows. For most cases where a client has a contractual agreement with the server, and the client can be trusted with resource owner password credentials, the Native Mobile Authorization Grant can be used. If the developer seeks to allow third-parties access to an API through a native authorization grant, the proposed solution (See Chapter 5) can be used.

6.3 Further Work

While the current situation for OAuth provides best-practice implementations for API provides, the security aspects and consequences for native mobile authorization flows is at best fuzzy. More study is needed on Android security principles, especially with regards to OAuth, both from academia and the industry. OAuth 2.0 is here to stay, but attackers are on a daily basis finding vulnerabilities in the implementations of even the biggest API providers using OAuth 2.0.

Appendix A

Terminology

A.1 OAuth 2.0

The following definitions are key to the OAuth 2.0 Authorization Framework as defined in RFC6749 [7, Section 1.1]. The definitions are logically ordered based on dependencies to other terms.

Client
: An application making protected resource requests on behalf of the resource owner and with its authorization. The term "client" does not imply any particular implementation characteristics (e.g., whether the application executes on a server, a desktop, or other devices).

Resource Server
: The server hosting the protected resources, capable of accepting and responding to protected resource requests using access tokens.

Resource Owner
: An entity capable of granting access to a protected resource. When the resource owner is a person, it is referred to as an end-user.

User Agent
: The redirection-capable agent, usually a browser, enabling the Resource Owner to communicate over the HTTP protocol.

Authorization Server
: The server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization.

Protected resource
: An access-restricted resource, owned by a Resource Owner.

Authorization Grant

a credential representing the resource owner's authorization (to access its protected resources) used by the client to obtain an access token.

Access Token

Credentials used to access protected resources. An access token is a string representing an authorization issued to the client. The string is usually opaque to the client. Tokens represent specific scopes and durations of access, granted by the resource owner, and enforced by the resource server and authorization server.[7, Section 1.4]

Refresh Token

Credentials used to obtain access tokens. Refresh tokens are issued to the client by the authorization server and are used to obtain a new access token when the current access token becomes invalid or expires, or to obtain additional access tokens with identical or narrower scope (access tokens may have a shorter lifetime and fewer permissions than authorized by the resource owner).[7, Section 1.5]

Client Credentials

Credentials used to identify a client to the authorization and resource servers. These credentials are optional, but recommended for many authorization flows.

Bibliography

[1] Google. App framework, Apr 2013. URL `http://developer.android.com/about/versions/index.html`.

[2] Google. Platform version distribution in market. accessed 2013-04-03, Apr 2013. URL `http://developer.android.com/about/dashboards/index.html`.

[3] Jeff Six. *Application Security for the Android Platform*. O'Reilly, 2011.

[4] Dag-Inge Aas. Authorization solutions on the internet. 2012.

[5] Google. Android developers, Apr 2013. URL `http://developer.android.com/`.

[6] Eran D. Hammer. Oauth history, July 2011. URL `http://hueniverse.com/oauth/guide/history/`.

[7] OAuth Working Group. "rfc6749 - the oauth 2.0 authorization framework". *RFC6749, Online, Accessed 2012-10-16*, 2012.

[8] Naitik Shah. Developer roadmap update: Moving to oauth 2.0 + https, May 2011. URL `http://dickhardt.org/2012/10/oauth-2-0/`.

[9] Lisa Dusseault Racha Dhamija. "the seven flaws of identity management". *Online, Accessed 2012-09-17*, 2012.

[10] Eran D. Hammer. Oauth 2.0 and the road to hell, July 2012. URL `http://hueniverse.com/2012/07/oauth-2-0-and-the-road-to-hell/`.

[11] Eran D. Hammer. #fuckoauth @realtimeconf, November 2012. URL `http://hueniverse.com/2012/11/fuckoauth-realtimeconf/`.

[12] Twitter. Changes coming in version 1.1 of twitter api, Aug 2012. URL `https://dev.twitter.com/blog/changes-coming-to-twitter-api`.

[13] re4k. Consumer keys of official twitter clients, Mar 2013. URL `https://gist.github.com/re4k/3878505`.

[14] Navid Ranjbar and Abdinejadi Mahdi. Authentication and authorization for mobile devices. 2012.

[15] Richard T. Watson Jane Webster. "analyzing the past to prepare for the future: writing a literature review". *MIS Quarterly*, 2006.

[16] Anders Kofod-Petersen. How to do a structured literature review in computer science, October 2012. URL `http://research.idi.ntnu.no/aimasters/files/SLR_HowTo.pdf`.

[17] Jared L Howland, Thomas C Wright, Rebecca A Boughan, and Brian C Roberts. How scholarly is google scholar? a comparison to library databases. *College & Research Libraries*, 70(3):227–234, 2009.

[18] John J Meier and Thomas W Conkling. Google scholar's coverage of the engineering literature: An empirical study. *The Journal of Academic Librarianship*, 34(3):196–201, 2008.

[19] Mohammad Nauman, Sohail Khan, Abu Talib Othman, Najeeb Ur Rehman, et al. Poauth: privacy-aware open authorization for native apps on smartphone platforms. In *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*, page 60. ACM, 2012.

[20] Nimish Radia, Martin Svensson, Kristoffer Gronowski, Bo Xing, and Andrew Ton. Method and system for conducting a monetary transaction using a mobile communication device, September 5 2012. EP Patent 2,495,695.

[21] Siham Rhermini. *Identity, Access Management and Single Sign-On Web-based Solutions*. PhD thesis, KTH, 2012.

[22] Gustaf Andersson and Fredrik Andersson. *Android Environment Security*. PhD thesis, LNU, 2012.

[23] Facebook. Getting started with the facebook sdk for android, Apr 2013. URL `http://developers.facebook.com/docs/getting-started/facebook-sdk-for-android/3.0/`.

[24] Google. About two-factor authentication, Apr 2013. URL `http://support.google.com/accounts/bin/answer.py?hl=en&answer=180744`.

[25] Twitter. xauth, Sept 2012. URL `https://dev.twitter.com/docs/oauth/xauth`.

[26] Google. Creating a custom account type, Apr 2013. URL `http://developer.android.com/training/id-auth/custom_auth.html`.

[27] Jeff Six. Stack overflow: Jeff six's response to "which is the safest way to include a pair of key (public/private) in a apk", Feb 2012. URL `http://stackoverflow.com/questions/9179066/which-is-the-safest-way-to-include-a-pair-of-key-public-private-in-a-apk/9179269#9179269`.

[28] Google. Getting started with the google+ platform for android, Apr 2013. URL `https://developers.google.com/+/mobile/android/getting-started`.