EXPLORING VARIED APPROACHES FOR COUNTERING THE PRIVACY AND SECURITY RISKS OF THIRD-PARTY MOBILE APPLICATIONS

by

Fadi Mohsen

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Computing and Information Systems

Charlotte

2016

ABSTRACT

FADI MOHSEN. Exploring varied approaches for countering the privacy and security risks of third-party mobile applications. (Under the direction of DR. MOHAMED SHEHAB)

Third-party mobile applications have become an essential component of modern ecosystems such as Android and iOS. The number of these applications has grown tremendously in recent years, for instance, the number of Android applications in the market is 1.8 million. In most cases, third-party applications need to access restricted resources on these platforms, thus, platform providers have deployed access control mechanisms to control the privileges these applications can obtain. Nonetheless, malicious applications continue to find their ways to gain privileged access to users data and profiles. The success of these malicious attacks depends largely on the efficiency of the access control mechanisms and users awareness and comprehension. In this dissertation, we investigate the security and privacy risks of third-party applications in Android system by identifying a number of vulnerabilities caused by the limitations of the Android permission access control system and bad implementation choices for a popular authorization standard called OAuth. We show how malicious applications can exploit these vulnerabilities. Then, a stand-alone and an integrated frameworks are proposed to detect and offer solutions for the discovered vulnerabilities. Moreover, in-lab and online user studies are conducted to study users awareness and comprehension to the vulnerabilities and the countermeasures.

The OAuth-WebView implementation is the most widely used approach despite explicit warnings to the developers of its security and privacy risks. OAuth-WebView in mobile applications is extensively studied in this dissertation for any security and privacy violations. The dissertation investigates another threat that is caused by third-party mobile applications, the Keylogging threat in Android. Keyloggeing threat has been reasonably studied in the computer systems but poorly covered in the domain of mobile operating systems. Android took the lead among the other mobile operating systems in allowing developers to build custom third-party keyboards to replace the stock Android keyboard. This opened the door for malicious developers to create keyloggers for the purpose of spying and/or fishing for users sensitive data. A malicious developer may build a keylogger from scratch or utilize an existing keyboard. Users may unknowingly install keyloggers off the online markets or may use a keylogger that a malicious user with physical access has installed on their devices. The third and last problem of this dissertation is Android Broadcast Receivers of system actions. Android provides finer-grained security features through a permission mechanism that puts limitations on the resources that each application can access. Upon installing a new Android application, a user is prompted to grant it a set of permissions. There are two typical assumptions made regarding permissions and mobile application security and privacy. The first one is that malicious applications need to retain many permissions. Secondly, mobile devices users assume that installed applications do not access data if they are not in the foreground. The dissertation seek to answer the following research questions:

- What are the OAuth in-app implementation choices that would put mobile users' privacy at risk?

- How to measure the mobile users' comprehension to OAuth-Embedded apps and what kind of security cue designs can be used to alert them of any OAuth related attacks?
- Are there any other components that must be taken into account to determine the malicious behavior of Android apps besides possessing dangerous permissions?
- How effective embedding security tips in Android applications can be in enhancing users' awareness?

For the purpose of answering the above questions, (1) The different OAuth implementations approaches in mobile applications adopted by popular resource providers are identified and possible attacks are being demonstrated. We summarize the OAuth implementation choices made by the service providers in their SDKs (Software Development Kits) and by developers in their OAuth-Embedded mobile applications. (2) In-lab and online user studies are conducted to evaluate users' awareness and comprehension to the OAuth-Embedded mobile applications and the possible attacks. (3) New security cue designs for WebView-based mobile applications are proposed and evaluated on observability, understandability and affectivity aspects. After realizing the problem of OAuth implementations in mobile applications (4) a stand-alone application-based solution called *OAuthManager* is proposed and a prototype is implemented. The solution is based on the concept of privilege separation and does not require high overhead. However, the solution mandates that both developers and service providers change their implementations to work with the proposed solution. Thus,

we introduce *SecureOAuth*, a whitelist access control protection framework for the Android platform. SecureOAuth is composed of: Android library modifications, service creation, and system app creation. A prototype of the SecureOAuth framework is implemented and evaluated on performance and memory overhead. The framework hardens the OAuth-WebView implementation with bounded overhead while keeping the user's involvement to minimum. Moreover, the framework requires no implementations' changes and assumes strong attacking assumptions. (5) An analysis study on a set of third-party Android keyboards, collected from the market, is conducted. The number and type of permissions that are requested by these keyboards make them potential victims for the keylogger attack. (6) The users and keyboard developers roles in increasing/decreasing the chance of successful keylogger attack is also studied. The study shows that keylogging threat is of high probability due to the current security configurations and users and developers choices. Moreover, the study shoes that the risk can be reduced by educating the users and by adopting new development approaches.As far as the broadcast receivers, the evolution of Android broadcast actions is studied, an attack scenario that made possible by the broadcast receivers is demonstrated, and a large dataset of benign and malicious Android applications is analyzed for the receivers usage pattern.

ACKNOWLEDGMENTS

The last five years would have overwhelmed me had it not been for all of the help and support i received from family, friends and colleagues. My deepest gratitude is to my advisor, Dr. Mohamed Shehab who has been a tremendous mentor for me. I have been amazingly fortunate to have an advisor who helped me grow as a researcher and an academic. Dr. Shehab's guidance has been significant. I would also like to thank my committee members, Dr. Bill Chu, Dr. Hadzikadic Mirsad, and Dr. Weichao Wang whose input and feedback during my proposal were insightful and valuable. I'd also like to thank Dr. Cem Saydam for his role during my proposal defense. Thank you all for your prompt responses to my emails to schedule the dissertation proposal and final defense. I would like also to thank Dr. William Tolone and Dr.Mary Lou, their patience and support helped me overcome a crisis situation and finish my Ph.D.

TABLE OF CONTENTS

LIST OF FIGURES xiv

LIST OF TABLES xviii

CHAPTER 1: INTRODUCTION 1

1.1. Open Standard for Authorization (OAuth) 5

1.1.1. OAuth and Mobile Applications 6

1.1.2. The Security Problems of OAuth 7

1.2. Keylogger Threat 7

1.2.1. Third-party Keyboards 8

1.2.2. Privacy Risks 8

1.3. Android Broadcast Receiver 9

1.4. Research Hypotheses and Contributions 9

CHAPTER 2: BACKGROUND 15

2.1. Mobile Third-party Applications 15

2.1.1. Android Operating System 15

2.1.2. Android Security 15

2.1.3. Android Permissions 16

2.2. Open Authorization (OAuth) 17

2.2.1. The Flows of OAuth 18

2.2.2. Embedded Web Browser Component 22

2.2.3. System Native Browser 24

2.2.4. Resource Provider Native App 26

2.3. Third-party Keyboards 28

2.3.1. Input Method Editor 29

2.3.2. Third-party IME Risks 30

2.3.3. The Privacy Risks of Risky IME 30

CHAPTER 3: THE PROBLEM OF IMPLEMENTING OAUTH IN MOBILE APPLICATIONS 32

3.1. Attacks Against OAuth Mobile App Implementations 32

3.1.1. Attacks against System Native Browser 32

3.1.2. Attacks Against OAuth Embedded Web Browser 33

3.2. SDKs and Apps Analysis 36

3.3. Threat Model 39

3.4. Adversary Model 40

3.5. Related Work 41

CHAPTER 4: PROPOSED SOLUTIONS 44

4.1. Potential Solutions 44

4.1.1. Byte-code Re-Writing 44

4.1.2. Moving Target Defense 45

4.2. Proposed Solution 1 : Privilege Separation - OAuthManager 46

4.2.1. Solution Details 46

4.2.2. Performance 48

4.2.3. Response time 48

4.2.4. Memory overhead 49

4.2.5. Security Analysis 50

4.3. Proposed Solution 2: Refactoring the WebView - SecureOAuth 50

4.3.1. Proposal 50

4.3.2. Our Proposal: SecureOAuth 51

4.3.3. Implementation Details 52

4.3.4. Chromium Library Modifications 53

4.3.5. SecureOAuth: System Application 54

4.3.6. SecureOAuthService: System Service 55

4.3.7. Evaluation 56

4.3.8. Processing and Memory Overhead 57

4.3.9. Security Analysis 59

CHAPTER 5: Security Cue Design User Studies 63

5.1. In-Lab User Study 63

5.1.1. User Study Description 67

5.1.2. User Study Design 67

5.1.3. Participants 70

5.1.4. User Study Results 70

5.2. Online User Study - Security Cue Designs for OAuth-WebView Attacks 86

5.2.1. Study Description 86

5.2.2. Participants 88

5.2.3. Security Cue Designs 88

5.2.4. Authentication Page - Security Cue Designs 88

5.2.5. Authentication Page - Survey 91

5.2.6. Participants Demography and Usage Pattern 98

5.2.7. The effect of the Security Cues on Participants Decisions 100

5.2.8. Comparing the Security Cues: Understandability and Noticeability 104

5.2.9. Security Cue Design for the Authorization Page 106

5.2.10. Authorization Page - Survey Questions 108

5.2.11. Comparing the Security Cues - Authorization Page: Understandability and Noticeability 109

5.2.12. Users' Privacy Assumptions: Web Applications Vs. Mobile Applications 113

CHAPTER 6: KEYLOGGER THREAT IN ANDROID 119

6.1. Keylogger Threat 119

6.1.1. Privacy Risks 121

6.1.2. Risky IME 121

6.1.3. Vulnerable Device 122

6.2. Empirical Study 123

6.2.1. Permissions Analysis 123

6.2.2. Popularity 124

6.3. User Study 126

6.3.1. Recruiting Participants 127

6.3.2. Number of Participants 127

6.3.3. Embedded Survey 128

6.4. Analysis 128

6.4.1. Participants Demography 128

6.4.2. Automatically Collected Data 130

6.4.3. Survey Data 132

CHAPTER 7: BROADCAST RECEIVERS 135

7.1. Android Broadcast Actions 136

7.1.1. The Danger with Broadcast Receivers 136

7.1.2. User Assumptions and Habits 138

7.1.3. Attack Model 138

7.1.4. Use Case Scenario 138

7.2. Related Work 139

7.3. Datasets Analysis 140

7.3.1. The Description of the Datasets 140

7.3.2. The Analysis Focus 140

7.3.3. The Analysis Results of The Benign Dataset 141

7.3.4. The Analysis Results of The Malware Dataset 141

7.3.5. Summary 148

7.4. Proposed Solutions 148

7.4.1. Android Package Installer 149

7.4.2. BroadcastsViewer Tool 150

CHAPTER 8: Conclusion 151

8.1. Dissertation Contributions 151

8.2. Limitations and Future Work 153

REFERENCES 155

LIST OF FIGURES

FIGURE 1: OAuth authorization code flow 19

FIGURE 2: OAuth embedded web client flow 24

FIGURE 3: OAuth native system browser 25

FIGURE 4: OAuth using resource provider app 28

FIGURE 5: Attack using the system native browser 34

FIGURE 6: Manipulating the authorization page 36

FIGURE 7: Facebook OAuth implementations 39

FIGURE 8: Dropbox OAuth implementations 40

FIGURE 9: The `SecureOAuth` system app is used to add common service provider URLs into the `ProtectedURLsDB`. 45

FIGURE 10: OAuth Manager running on a separate application and communicating with the client apps via the intents 46

FIGURE 11: Solution Architecture 51

FIGURE 12: SecureOAuthService family tree. 54

FIGURE 13: The difference in response time between a fresh image (UM) and a modified image (M) of the Android Kitkat version. 57

FIGURE 14: The performance overhead is being split between the different implementation's segments. 58

FIGURE 15: The Facebook authorization page before using our solution (left) and after (right). 59

FIGURE 16: The page that we used in testing our solution, the page before js injection (left) and after js injection (right) . 60

FIGURE 17: Using different warning elements on the Authorization page: In a, the red border is used. In b, the highlight on portion of the permissions and a toast message are used. 65

FIGURE 18: The first image shows a normal authorization page, the second image shows a manipulated authorization page where one permissions is been taken out. 66

FIGURE 19: Using different warning elements on the Authentication page: In a, the red border and the text ”Not Safe” are used. In b, red back ground for the input fields and the text ”Not Safe” are used. 68

FIGURE 20: The figure in a and b shows a normal authentication page and a hacked authentication page respectively. In b, the app reads user password and display it in a toast message. 69

FIGURE 21: Reluctance to Enter Credentials. 72

FIGURE 22: Noticing the Security Clues. 73

FIGURE 23: Understanding the Security Clues. 74

FIGURE 24: Participants‘ Assumptions with Regard to Who Generates the Authentication Page. 76

FIGURE 25: Participants‘ Assumptions with Regard to Who Toasts the Password. 77

FIGURE 26: Participants‘ Assumptions with Regard to Who Generates the Authorization Page. 80

FIGURE 27: Sharing credentials with Facebook only. 81

FIGURE 28: Sharing credentials with Facebook and App/Website. 83

FIGURE 29: App / Website can access content and data. 84

FIGURE 30: Design D uses a red line around the authentication page. 89

FIGURE 31: Design C toasts the password as the user enters it. 89

FIGURE 32: Design B uses the red sign (Traffic Light). 90

FIGURE 33: Design A uses red backgrounds for the input fields and a message). 90

FIGURE 34: SPSS Crosstabulation Results of the Logedin class. 101

FIGURE 35: SPSS Chi-Square Results of the Logedin class. 102

FIGURE 36: SPSS Crosstabulation Results of the Account class. 102

FIGURE 37: SPSS Chi-Square Results of the Account class. 102

FIGURE 38: Design C displays a red border around the authorization page. 106

FIGURE 39: Design B uses the red sign (Traffic Light). 107

FIGURE 40: Design A highlights parts of the permissions and display a toast/warning message). 107

FIGURE 41: SPSS Crosstabulation Results of the Authorize class. 110

FIGURE 42: SPSS Chi-Square Results of the Authorize class. 110

FIGURE 43: The download frequency for the collected third-party keyboards 123

FIGURE 44: The frequency for the number of permissions requested by collected apps 124

FIGURE 45: The 17 most requested permissions and the percentage of third-party keyboards that request them 125

FIGURE 46: The rating frequency for the collected third-party keyboards 125

FIGURE 47: Among the studied set, 15% requested 0 permission, 85% requested one or more permission 126

FIGURE 48: The number and frequency of installing third-party application by the participants. 128

FIGURE 49: The security configurations for participants' Android devices. 129

FIGURE 50: Measuring participants' security awareness by comparing device real configurations with their answers (Rooting). 130

FIGURE 51: Measuring participants' security awareness by comparing device real configurations with their answers (Screen Lock). 130

FIGURE 52: The distribution of different Apps resources. 132

FIGURE 53: Participants responses to the security questions. 134

FIGURE 54: Android Devices Usage Behavior. 134

FIGURE 55: The evolution of Android broadcast actions over the 19 releases. 136

FIGURE 56: The change in the number of Android broadcast actions over the 19 APIs. 137

FIGURE 57: BroadcastsViewer Tool for Detecting and Displaying Registered System Broadcast Actions. 149

LIST OF TABLES

TABLE 1: OAuth SDKs and authentication methods 32

TABLE 2: Comparison of response time (milliseconds) 49

TABLE 3: Comparison of memory consumption (kB) 50

TABLE 4: Memory Overhead. 59

TABLE 5: The different attacker's skill levels that we considered for the WebView-based attacks. 60

TABLE 6: Comparing SecureOAuth to the other solutions. 62

TABLE 7: The eight scenarios that are used in designing the study testing applications. 64

TABLE 8: The seven testing applications used in our user study. 64

TABLE 9: Pairwise Comparisons Results - Reluctance to Enter Credentials. 72

TABLE 10: Pairwise Comparisons Results - Noticing the Security Clues. 74

TABLE 11: Pairwise Comparisons Results - Understanding the Security Clues. 75

TABLE 12: Pairwise Comparisons Results - Who Generates Authentication Page. 75

TABLE 13: Pairwise Comparisons Results - Who Toasts Password. 78

TABLE 14: Pairwise Comparisons Results - Who Generates Authorization Page. 79

TABLE 15: Pairwise Comparisons Results - Sharing with Facebook Only. 82

TABLE 16: Pairwise Comparisons Results - Sharing with Facebook and App/Website. 82

TABLE 17: Pairwise Comparisons Results - App / Website can access content and data. 84

TABLE 18: Participants Demography 99

TABLE 19: Participants Mobile Phone Usage Pattern 100

TABLE 20: The reasons why participants chose not to login at all. 103

TABLE 21: The reasons why participants chose the study Facebook account to login. 103

TABLE 22: The reasons why participants chose their own Facebook account to login. 104

TABLE 23: Chi-Square Test Results of the Visibility Variable - Frequencies Table. 104

TABLE 24: Chi-Square Test Results of the Visibility Variable - Statistics Table. 105

TABLE 25: Chi-Square Test Results of the Understandable Variable - Frequencies Table. 105

TABLE 26: Chi-Square Test Results of the Understandable Variable - Statistics Table. 105

TABLE 27: The reasons why participants chose to authorize the application on their/Study Facebook account. 110

TABLE 28: The reasons why participants chose to authorize the application on their/Study Facebook account. 111

TABLE 29: (Authorization) Chi-Square Test Results of the Understandable Variable - Frequencies Table. 111

TABLE 30: Authorization) Chi-Square Test Results of the Understandable Variable - Statistics Table. 112

TABLE 31: (Authorization) Chi-Square Test Results of the Visibility Variable - Frequencies Table. 112

TABLE 32: (Authorization) Chi-Square Test Results of the Visibility Variable - Statistics Table. 112

TABLE 33: The descriptive statistics for participants' responses to Question 1. 114

TABLE 34: The Symmetric measures for participants' responses to Question 1. 115

TABLE 35: The descriptive statistics for participants' responses to Question 3. 115

TABLE 36: The Symmetric measures for participants' responses to Question 3. 116

TABLE 37: The descriptive statistics for participants' responses to Question 4. 116

TABLE 38: The Symmetric measures for participants' responses to Question 4. 117

TABLE 39: The descriptive statistics for participants' responses to Question 5. 117

TABLE 40: The Symmetric measures for participants' responses to Question 5. 117

TABLE 41: The descriptive statistics for participants' responses to Question 6. 118

TABLE 42: The Symmetric measures for participants' responses to Question 6. 118

TABLE 43: Participants Demography 127

TABLE 44: The post survey results of 54 participants. 134

TABLE 45: The number of receivers found for the system actions in the benign dataset (Top most received). 142

TABLE 46: The number of receivers found for the system actions in the benign dataset (Moderate). 143

TABLE 47: The number of receivers of the system actions found in the benign dataset (Low). 144

TABLE 48: The number of receivers of the system actions found in the benign dataset (Very Low). 145

TABLE 49: The number of receivers found for the system actions in the malware dataset (Top most received). 146

TABLE 50: The number of receivers found for the system actions in the malware dataset (moderate). 146

TABLE 51: The number of receivers found for the system actions in the malware dataset (low). 147

TABLE 52: A summary of the dataset that we analyzed for the Broadcast receivers information. 148

CHAPTER 1: INTRODUCTION

Smartphones and tablets are continuously transforming the lives of millions of people, which is mainly supported by the astonishing growth in mobile applications (apps). For example, in the first quarter of 2013, Apple customers have downloaded more than 40 billion apps and developers have created more than 775,000 apps. In the first quarter of 2016, the number of Android applications in the market in the first quarter of 2016 was 1.8 million generating billions in revenue. Developing mobile apps is becoming more accessible to developers, and plenty of development resources and support are available. As a result, millions of applications are now available for customers to download some of which are paid others are offered for free. Moreover, these applications are so diverse, one could almost find an application for every aspect in life, such as, applications for mail, chat, text, bank, social web sites, etc. The majority of these applications usually request to access the phones' resources such as phone book, camera, and sensors, etc. Mobile phones' operating systems have adopted mechanisms to control the level of access these applications can obtain. For example, Android [1] and Apple [2] provide applications with finer-grained security features through a "permission" mechanism that enforces restrictions on the access request that a particular application can perform. Despite of the this mechanism, many applications have managed to compromise the privacy of Android users by taking advantage of their lack of understanding and awareness of the permissions [21] and some limitations of this mechanism [22]. Previous works on Android permissions were geared towards helping users pay more attention to the applications requests [21], detecting and preventing permissions sharing and hacking across applications [22, 68, 62], tracking information leakage originated by possessing dangerous permissions [72], and detecting malwares based on the permissions the possess [33]. The vast majority of these works rely on the dangerous permissions and sensitive information to determine the benign and the malicious cases and on the sensitive information to determine the leakage cases .The risk is even higher in the case of some applications that require users authorization to access remote servers on their behalf. For instance, the social networking sites (e.g. Facebook and Twitter) and cloud storage services (e.g. Dropbox and Box) allow their users to connect to their services via mobile applications. Popular social networking services are also now being used as identity providers by other web sites and services. Secure authentication and authorization standards are adopted to enable authenticated users to grant authorizations to mobile apps to access the users' resources that are stored in their devices and/or remote servers. The Open Standard for Authorization (OAuth) is an example of an authorization standard. Many service providers such as Google, Github, and Facebook have OAuth endpoints to support third-party mobile applications. Throughout this dissertation, the term third-party applications will refer specifically to those apps that are developed by anyone other than the platform vendor. For instance, an Android application that is developed by any freelance or a company other than Google is considered a third-party application. In this dissertation three problems related to third-party mobile applications are going to be investigated (1) secure OAuth implementations (2) keylogger threat and (3) Broadcast receivers.

The Open Standard for Authorization (OAuth) [65] is widely used by the industry in comparison to similar standards such as Google AuthSub, Microsoft Live ID, OpenID, and Yahoo BBAuth. Many service providers such as Google, Github, and Facebook have OAuth endpoints to support the third-party applications that are installed on mobile devices. The service providers who adopted OAuth have included its implementation in their Software Development Kits (SDKs). The mobile apps are using OAuth/SDKs to gain access to users' resources on these services [3]. As part of the OAuth implementation [69], these applications must either have access to the system browser, or they must have a browser embedded as a web view. In Android, the embedded browser is called *WebView* [10]. Previous researches have discussed the risk of using WebView and its APIs on web's security in general and OAuth in particular [48, 12]. Using the WebView as part of OAuth implementations in mobile applications is unsafe and vulnerable to JavaScript injection attacks [3].

Unlike the majority of mobile operating systems, Android was the first mobile operating system to allow users to install third-party keyboards. To be more precise, Android started supporting the installation of third-party keyboards since version 1.5 (Known for its code name, Cupcake [4]). Users of other mobile operating systems, such as iOS [77], have a long standing criticism of iOS for not allowing the installation of third-party keyboards. However, with the introduction of iOS 8 –the biggest iOS release ever according to Apple– third-party keyboards were finally allowed on both iPhone and iPad. Third-party keyboards can be seen as an opportunity to enjoy new features, security experts on the other hand see them as a potential threat to

users‘ privacy. The minute a user installs and activates a third-party keyboard, all of her keystrokes might be leaked or misused. Thus, it is the responsibility of mobile operating systems, third-party keyboard providers, and mobile users to ensure the safety of these keyboards. A malicious third-party keyboard once activated can leak any data the end user enters into his or her device. The data can be text messages, emails, credentials, phone numbers, social security numbers, credit card numbers, etc. Any of these data types can be misused to achieve damage, profit or both. For example, an attacker may use stolen text messages to blackmail their owners or use stolen social security numbers to commit a fraud. Thus, the severity of any keylogger attack depends largely on the device vulnerability level and user usage behavior. For instance, a keylogger attack against a rooted Android device that is used by an end user to do banking and economical activities may result in a severe damage or loss.
A broadcast receiver is an Android component that enables applications to register for system or application events or actions(e.g. receive call, receive message). Once an event occurs, Android runtime notifies all registered receivers of that particular action. A broadcast is a message that any application can receive. The Android system delivers numerous broadcasts for system events, such as when an Internet connection is enabled or a new SMS arrives. While broadcast receivers are considered a useful feature by developers, users' experience and privacy can be affected negatively by them. Therefore, investigating Android broadcast receivers is vital. The next sections of this chapter introduce the open standard for authorization (OAuth) and its security problems, Keylogger threat in Android, and the privacy risks of Android System Broadcasts receivers.

1.1 Open Standard for Authorization (OAuth)

Open standards such as Open Authorization (OAuth), allow the resource owner (user) to grant permissions to a third-party (mobile app) access to their information hosted on a resource provider (Facebook). With the OAuth technology, the users are no longer required to share their credentials to third party apps in order to grant them authorizations. In addition, OAuth allows different access granularity, where users are able to grant access to specific resources, and there are provisions for revoking access at any time. Other related authorization approaches include Google AuthSub [28], Microsoft Live ID [51], OpenID [14], and Yahoo BBAuth [78]. OAuth is the most adopted with over one billion OAuth-based user accounts supported by the major online service providers. Major services providers offer software development kits (SDK) that can be included in apps to enable the apps to be seamlessly integrated with their services. However, if other parties choose to develop their owns libraries they are required to follow the standards required by the service providers. Through the SDKS, service providers offer their own implementations of the authentication and authorization protocols. For example, the different OAuth implementations adopted by popular mobile SDKs vary in their security assumptions and guarantees. Several mobile SDKs rely on embedded web components to execute the OAuth authentication and authorization stages, which does not provide the required isolation and can easily be exploited by malicious apps.

1.1.1 OAuth and Mobile Applications

Similar to web and desktop applications, native mobile applications (apps) in many usage scenarios require access to user resources hosted on a resource server. For example, a photo sharing application that enables users to take photos with their smart phone's camera, tag photos, create albums and share them with their Facebook friends. Native mobile apps that are installed and executed on mobile devices are considered public clients and utilize special kind of user-agents besides the system browser, called the *WebView.* Thus, OAuth authorization flow steps and the logistics beforehand differ accordingly.

For instance, in case of web applications, the authorization server grants them credentials and passwords for the purpose of client authentication. The web applications has to keep these information confidential and safe. In case of native applications, the authorization server can't use the same means (passwords and credentials) since these clients aren't capable of keeping such data confidential. Instead, the authorization server require these clients to register redirection URI and/or asking the resource owner to approve identity. Another advantage of client registration and resource owner engagement is to prevent from client impersonation. Its when a malicious client impersonate another client to obtain access to protected resources if the client fails or unable to keep its credentials confidential. So when registering the client, the client developer shall specify the client type, provide its client redirections URIs, and some other information needed by the authorization server. Overall, the sequence for installed applications is different than Web Server in:

- Upon application registration, client type must be set to installed application. This results in a different value for the redirect_uri parameter,
- Both client_id and client_secret obtained while registering will be included in the source code, thus, the client_secret is not literally a secret,
- The authorization code is returned differently.

1.1.2 The Security Problems of OAuth

The majority of mobile apps that are built using provider SDKs have serious security flaws, some of which are OAuth related [65]. Using OAuth for mobile applications was proven to be unsafe due to flaws in the OAuth specifications and OAuth implementations [69]. For example, recent studies have found that the implementations of OAuth in mobile applications could put users privacy at risk [3]. Some of these implementations' vulnerabilities have been linked to developers misunderstandings of OAuth specifications [10]; others are related to the implementations' choices. For instance, using WebView to display the authentication and authorization pages could result in different kinds of attacks [3]. The WebView itself has been found vulnerable to different kinds of attacks [48]. Chin and Wanger conducted a systematic study [12] on a data set of 864 applications and foud that 70% of these apps contain WebViews. Among the 70%, they found that over 20% of applications have the potential to give websites access to code and 54% allow a user to navigate to malicious JavaScript that could access application‘s code.

1.2 Keylogger Threat

In this section, we talk about the Keylogger threat in Android system.

1.2.1 Third-party Keyboards

Unlike the majority of mobile operating systems, Android was the first mobile operating system to allow users to install third-party keyboards. To be more precise, Android started supporting the installation of third-party keyboards since version 1.5 (Known for its code name, Cupcake [4]). Users of other mobile operating systems, such as iOS [77], have a long standing criticism of iOS for not allowing the installation of third-party keyboards. However, with the introduction of iOS 8 –the biggest iOS release ever according to Apple– third-party keyboards were finally allowed on both iPhone and iPad. Third-party keyboards can be seen as an opportunity to enjoy new features, security experts on the other hand see them as a potential threat to users‘ privacy. The minute a user installs and activates a third-party keyboard, all of her keystrokes might be leaked or misused. Thus, it is the responsibility of mobile operating systems, third-party keyboard providers, and mobile users to ensure the safety of these keyboards.

1.2.2 Privacy Risks

A malicious third-party keyboard once activated can leak any data the end user enters into his or her device. The data can be text messages, emails, credentials, phone numbers, social security numbers, credit card numbers, etc. Any of these data types can be misused to achieve damage, profit or both. For example, an attacker may use stolen text messages to blackmail their owners or use stolen social security numbers to commit a fraud. Thus, the severity of any keylogger attack depends largely on the device vulnerability level and user usage behavior. For instance, a keylogger attack against a rooted Android device that is used by an end user to do banking and economical activities may result in a severe damage or loss. For this reason, our user study is focused towards capturing and collecting users usage behaviors and possible vulnerabilities.

1.3 Android Broadcast Receiver

A broadcast receiver is an Android component that enables applications to register for system or application events or actions(e.g. receive call, receive message). Once an event occurs, Android runtime notifies all registered receivers of that particular action. A broadcast is a message that any application can receive. The Android system delivers numerous broadcasts for system events, such as when an Internet connection is enabled or a new SMS arrives. While broadcast receivers are considered a useful feature by developers, users' experience and privacy can be affected negatively by them. Therefore, investigating Android broadcast receivers is vital.

1.4 Research Hypotheses and Contributions

Previous works on Android Broadcast Receivers [11] [46] looked at Broadcast receivers components as mailboxes for exchanging messages between applications. Thus, the focus was on securing this communication channel from being hijacked or misused. There is no previous works that talk about this component as a communication channel between the Android system and third-party applications or looked at the type of the information that the system broadcasts have. For the keylogging threat, our paper [54] was the first work that points to the privacy risks of third-party keyboards in Android and how can be converted into a keyloggers. Following this work, there has been some works on Android keyloggers [23] that followed our research method.

Our contributions in the OAuth problem was unique in that it was the first work that pointed out the privacy risks of using the WebView , which was found by other researchers to be vulnerable [48], in implementing OAuth in mobile applications. The dissertation investigates the following hypotheses within the contexts of the above three main problems.

- Implementing OAuth in smartphone's applications may jeopardize users' privacy.
- The majority of service providers and application developers choose functionality over security.
- Security cue designs for WebView based Android applications do not exist and creating ones is a necessity and will be valuable to the end users.
- Solving the WebView-OAuth security problems if exist can be very challenging.
- The likelihood of Keylogging threat in Android can be high and the consequences can be severe.
- Embedding security tips in Android applications can enhance users' awareness.
- There exit privacy attacks that do not require dangerous permissions but other components might be needed.

For the purpose of investigating the above hypotheses, we (1) analyze the different OAuth implementations adopted by the SDKs of the popular resource providers on smartphones and demonstrate possible attacks on most OAuth implementations. By analyzing source codes of popular Android apps collected from online markets and the SDKs of popular service providers, we summarize the trends followed by the service providers and the OAuth development choices made by application developers. We then demonstrate various kinds of OAuth-based attacks against WebView and native browser approaches. (2) We conducted an in-lab user study and online large scale user study to evaluate users‘ security and privacy assumptions of third-party mobile applications and compare them to web applications. The users studies also investigated the effectiveness of different security cue designs that could be used to alert the end users of attacks against the integrity of the OAuth's authorization pages and the confidentiality of the data of its authentication pages. The studies also tend to investigate users' separation awareness between the mobile app context and provider context, in cases where the mobile apps communicate with online service providers (e.g. Facebook, Twitter, and Dropbox). The separation includes the authorization page's contents and the data being entered (e.g. credentials). Our empirical study showed that most of investigated apps and SDKs are vulnerable to numerous kind of attacks. In addition, our user studies results showed that smartphones users lack the separation awareness between mobile apps and service providers. The lack of understanding could lead into trusting a malicious content that could result in compromising users privacy. Also, the results show that users have less trust in third-party mobile applications. Based on our previous findings we realize that there is a necessity to harden OAuth-WebView against the recognized attacks. (3) We first started by looking at stand-alone application-based solutions to the OAuth vulnerabilities in which a secure WebView is implemented in a separate application to provide a secure

OAuth flow in smartphones. The solution is based on the concept of privilege separation and does not require high overhead. However, the solution mandates that both developers and service providers change their implementations to work with the proposed solution. (4) We proposed a second solution, SecureOAuth, a whitelist access control protection framework for the Android platform. SecureOAuth is composed of: Android library modifications, service creation, and system app creation. We have implemented a prototype of the SecureOAuth framework and evaluated it on performance and memory overhead. We also showcase examples of security threats that this framework counters. The framework hardens the OAuth-WebView implementation with bounded overhead while keeping the users involvement to minimum. Moreover, the framework requires no implementations changes and assumes strong attacking assumptions.

The dissertation investigates another threat that is caused by third-party mobile applications, the Keylogging threat in Android. Keyloggeing threat has been reasonably studied in the computer systems but poorly covered in the domain of mobile operating systems. Android took the lead among the other mobile operating systems in allowing developers to build custom third-party keyboards to replace the stock Android keyboard. This opened the door for malicious developers to create keyloggers for the purpose of spying and/or fishing for users sensitive data. A malicious developer may build a keylogger from scratch or utilize an existing keyboard. Users may unknowingly install keyloggers off the online markets or may use a keylogger that a malicious user with physical access has installed on their devices. (5) We conducted an analysis study on a set of third-party Android keyboards, collected from the market. The number and type of permissions that are requested by these keyboards make them potential victims for the keylogger attack. (6) We study users and keyboard developers roles in increasing/decreasing the chance of successful keylogger attack. In so doing, we developed an Android app, KBsChecker , and asked participants to install it on their devices. The app collects data off participants devices and prompts them to complete a survey. We also asked a number of developers to answer few questions with regard to their experience in building third-party keyboards. Our study showed that keylogging threat is of high probability due to the current security configurations and users and developers choices. Moreover, the study showed that the risk can be reduced by educating the users and by adopting new development approaches.

The third and last problem of this dissertation is Android Broadcast Receivers. Android provides finer-grained security features through a permission mechanism that puts limitations on the resources that each application can access. Upon installing a new Android application, a user is prompted to grant it a set of permissions. There are two typical assumptions made regarding permissions and mobile application security and privacy. The first one is that malicious applications need to retain much permission. Secondly, mobile devices users assume that installed applications do not access data if they are not in the foreground. In this dissertation, we show that malicious Android applications can still fulfill their objectives with minimum permissions and that they can access user data while in the background. This could happen with the help of another Android component, called broadcast receiver. (7) We study the evaluation of Android broadcast actions, demonstrate an attack scenario made possible by the broadcast receivers, and analyze thousands of malicious and benign

Android applications for their usage of the broadcast receivers. (8) Moreover, we developed an app to aid user identify the broadcasts receivers that are registered by the apps on their own Android devices called Broadcasts Viewer.

CHAPTER 2: BACKGROUND

2.1 Mobile Third-party Applications

2.1.1 Android Operating System

The Android operating system has been known for it‘s openness and support for numerous features. The numerous features have been inclemently introduced over the different Android releases. For instance, the Android 1.5 release known by its code, Cupcake, supported both the virtual and the physical keyboards. It also introduced the hooks that are needed by third-party app developers to build their own customized keyboards. These kind of hooks were not supported by other mobile operating systems, for instance, iOS was not allowing them till iOS 8. Android developers have been taking the advantage of the advanced features that are exposed through the platform for building innovative and useful applications for the end user. To ensure the privacy of users, data, and applications, while keeping the platform open, Android was designed with multi-layered security architecture.

2.1.2 Android Security

Android platform architecture is composed of four main layers: Linux kernel layer, libraries layer , application framework layer, and applications layer. Right on top of the device hardware sits the Linux kernel layer. The Linux kernel has been used in numerous platforms including Android as a secure multi-user operating system. It also includes the device drivers, for example, Wi-Fi, Bluetooth, and flash memory.

One of the basic virtue of the Linux kernel is isolating one user from the other. This property represents the cone stone of Android security system. Android considers each application as a different user, thus, each application is isolated from each other. The Library or the Middleware layer includes native libraries and runtime environment for Java applications, Dalvik Virtual Machine. Each Java application runs in its own virtual machine which represents the second layer of isolation between Android applications. The next layer is the application framework which contains classes and resources to assist with building Android applications. The last layer is the applications layer which entails the pre-installed applications and user-installed applications. Android devices are normally shipped with a set of pre-installed applications such as web browser, calendar, messaging, and keyboard. In addition to these applications, users may choose to install third-party applications off online app stores. For instance, a user may install third-party keyboard to replace the keyboard that was shipped with the device. However, third-party apps may pose security and privacy risks on Android devices and users data. Because of that, Android system regulates the access level or privileges each application can get using the permissions system.

2.1.3 Android Permissions

Android permissions system guarantees that an application must have permissions in order to make any API system calls. An API system call allows applications to gain access to system and/or user resources, for example, user‘s pictures or sending an SMS. Android mandates developers to explicitly list the permissions their applications require in the configuration file called AndroidManifest.xml. Users upon installing the applications they are prompted to agree to grant the displayed permissions otherwise the application would not be installed. There have been several works about Android permissions and their related risks. The danger comes from granting too many permissions to un-trusted applications that try to collect users‘ sensitive information off their devices. Another form of danger can result from granting a particular combination of permissions. In this work, we look at the risk that can result from a third-party keyboard possessing a combination of permissions.

In this section we discuss the basics of the Open Authorization (OAuth) flows and the state of the art of OAuth implementations in smartphones. We identify some implementation details that may open the gate to some potential attacks.

2.2 Open Authorization (OAuth)

Open Authorization (OAuth) was first introduced as an OpenID implementation which only authenticated users to a third party application allowing users to log in using an OpenID such as Twitter. OAuth 1.0 [43] addressed more than just authentication, it also allowed end users to authorize third party applications access to their online resources on a server without revealing their server credentials or password to those applications. The resource server and the client (third party application) share a token which is used by the client to access APIs provided by the server. OAuth provided a solution to problems pertaining to old authorization frameworks giving the resource owner control on how much data a third party can access, when the access is granted, and how long it is granted. For example, a photo sharing website such as Flickr is a resource server, and a photo printing service such as Snapfish is the client. The user might authorize the printing service read-only access to a subset of the photos posted on Flickr. In OAuth 1.0, using a secure communication channel was not a requirement and each API call is signed by the client and the signature is verified by the server with each API call. OAuth 2.0 [44] requires the use of SSL in all communication performed to generate the access token. OAuth 2.0 provides a clear specification of the protocol and greatly simplifies the implementation which has led to its adoption in several popular resource serving websites such as Twitter, Facebook, and Google.

2.2.1 The Flows of OAuth

OAuth provides several flows to accommodate different usage scenarios varying from web server to client based applications. In this paper, we focus on client based OAuth flow, as it is the applicable flow followed by mobile apps. OAuth defines four main roles or parties that are involved in the protocol flow, namely: *resource owner, resource server, authorization server, and client.* The *resource owner* is the entity which owns the resources and is capable of granting access to its protected resources. The *resource server* hosts protected resources and is capable of reading and writing protected resources when requested using access tokens. The *authorization server* issues access tokens to a client once the resource owner authenticates themselves and authorizes the client to use its protected resources. In several implementations, the resource and authorization servers are provisioned by the same entity. The *client* is the third party application which is requesting access to the protected resources by the resource owner. The client makes requests to the resource server on behalf of the resource owner using the access token it received from the authorization server.

Figure 1 describes the authorization code flow, which are the steps required for the client to acquire an access token.

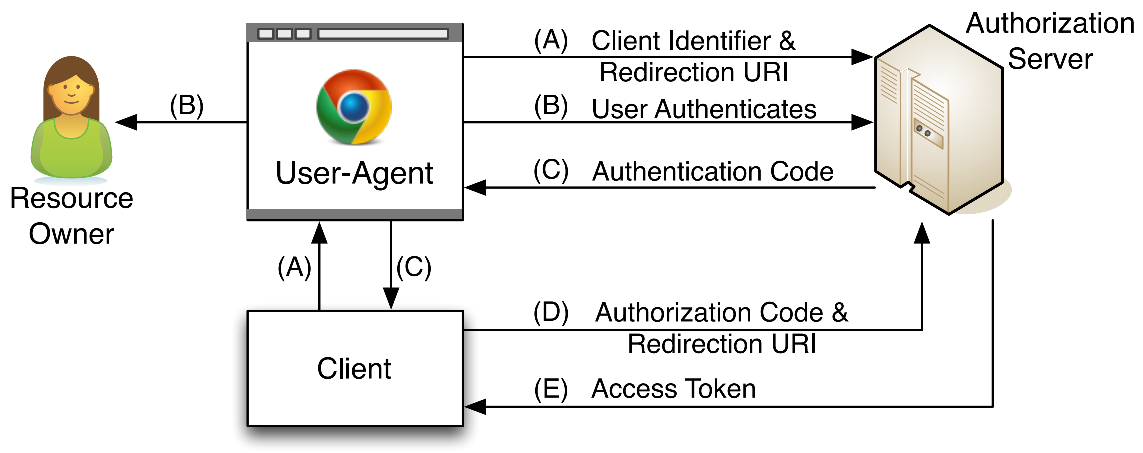


Figure 1: OAuth authorization code flow

- Step A: The client starts the authorization flow by directing the user-agent, which is typically a web browser, to the authorization server requesting authorization (Step A). The client includes its client identifier, requested permissions (scope), local state, and a redirection URI to which the authorization server will send the user-agent back once the access is granted (or denied).

- Step B: Through the user-agent, the authorization server authenticates the resource owner usually by requesting username/password. Then the resource owner is presented with access authorizations requested by the client and is asked to grant or deny the client's access request.

- Step C: Assuming the resource owner grants access, the authorization server redirects the user-agent back to the client using the redirection URI provided earlier (in the request or during client registration). The redirection URI includes an authorization code and any local state provided by the client earlier.

- Step D: Using the authorization code received in the previous step the client requests an access token from the authorization server. The client authenticates with the authorization server, and includes the redirection URI used to obtain the authorization code for verification.

- Step E: The authorization server authenticates the client, validates the authorization code, and ensures that the redirection URI received matches the URI used to redirect the client in Step (C). If valid, the authorization server responds back with an access token and, optionally, a refresh token.

OAuth defines several authorization grant types for clients to request an access token:

- Authorization Code: Follows the flow described in Figure 1, where the client is issued an authorization code prior to obtaining the access token.
- Implicit: Similar to the flow described in Figure 1, however an access token is returned directly in step (C) without the authorization code steps (D & E).
- Resource Owner Password Credentials: The client uses the resource owner password credentials to directly obtain an access token.
- Client Credentials: Used when the client is also the resource owner. The client credentials are used when the authorization scope is limited to the protected resources that are under the control of the client.

Different authorization grant types have different requirements, for example, the implicit grant is a simplified authorization code flow optimized for clients implemented in a browser using a scripting language such as JavaScript. The authorization grant types differ also in their security level. For instance, the authorization code grant has a security advantage over the implicit grant. The client in the authorization code grant has first to get an authorization code, then using that code it requests to obtain an access token. The client gets the access token directly without passing it through the resource owner's user-agent, potentially divulging it to others, including the resource owner. The implicit grant in the other hand does not have this step, instead it requests to obtain an access token directly. However, the implicit grant is more secure than the resource owner password credentials grant due to the fact that user credentials are not shared with the client. OAuth 2.0 defines two client types confidential and public clients. Confidential clients are expected to maintain the confidentiality of their client credentials, while the public clients are not expected to provide such confidentiality guarantees. The client classification is based on the authorization server's security criteria and considerations. For each client type, the authorization server provide different type of flows and provide guidance with regard to their registration. OAuth 2.0 specification is centered around the following client profiles:

- Web application: A confidential client running on a web server.
- User-agent-based application: A public client where the client code is retrieved from a web server and executed on the browser on a device used by the resource owner.
- Native application: A public client that is installed and run on a device used by the resource owner.

In the next subsection, we discuss the differences between web applications and native applications, then we talk about the difference between desktop applications and mobile applications in terms of user authentication techniques.

Several of the OAuth authorization flow steps require a user-agent, which is usually a web browser. On desktops, the web browser's isolation mechanisms, such as the same origin policy, provides the required separation between the user-agent, client and authorization server. The user-agent presents the resource owner with the authentication and authorization information, and the user-agent is used to redirect and pass tokens between the client and the authorization server. In mobile applications, the user agent is implemented using one of the following approaches:

- Using a web component embedded in the client app.
- Using the resource provider's native mobile app.
- Using the system's native browser.

Each user agent implementation offers different security and isolation guarantees.The details of each approach will be discussed in the following section.

2.2.2 Embedded Web Browser Component

The web browser component is a UI view component that can be embedded in a mobile app to display online contents within the hosting app. This component is available in the different mobile frameworks, WebView in Android platform, UIWebView in iOS, and WebBrowser in Windows Phone. The Android WebView uses the WebKit rendering engine to display web pages and includes methods to navigate forward and backward through a history, zoom in and out, and perform text searches. We will focus our discussion on the Android platform, however the following discussion is applicable to other platforms. The WebView is used to perform the role of the user-agent during the OAuth authorization code flow, particularly, it is used to present users with the required OAuth authentication and authorization pages on mobile the applications. The client app hosts a WebView as part of its' UI layout. The app can control the embedded WebView, for example it can load a specific url "`webview.loadUrl(URL)`", enable JavaScript in the WebView "`setJavaScriptEnabled(true)`", register event handlers to the WebView to respond to events and to monitor the WebView's activity "`onLoadResource()`" and "`onPageFinished()`", and inject a native (Java) object into the WebView to allow the object's method to be accessed from JavaScript "`addJavascriptInterface()`".

Figure 2, shows the OAuth flow for client apps that use an embedded web browser (WebView). The client code triggers the WebView to load the service provider's authentication page (Step A0), where the user is promoted to provide her username and password. The WebView also loads the authorization page (Step B), which displays the permissions requested by the application. Upon submission of the authorization page, the authorization server responds and the authentication code is sent as part of the response page title or content (Step C0). The client code has initially registered the event handler "`onPageFinished()`" to be notified when the WebView is done loading pages. The event handler is notified with the loading of the authentication response page, which enables the client code to retrieve the authentication code from the loaded page title or content (Step C1). The client code then uses the authentication code to retrieve the authentication token from the authorization server (Steps

D & E).

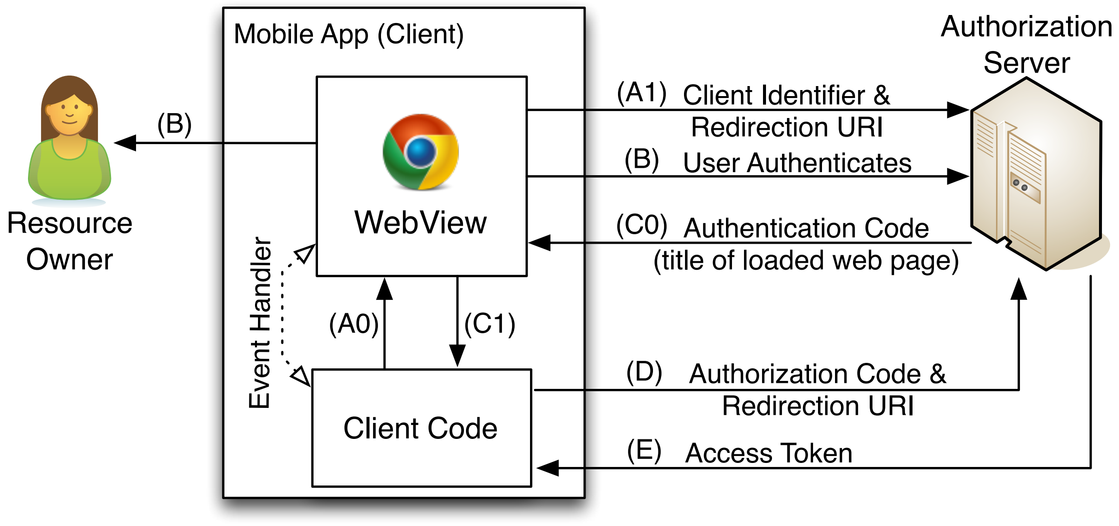


Figure 2: OAuth embedded web client flow

2.2.3 System Native Browser

Several service providers rely on the system native web browser for both the user authentication and the app authorization, where the native browser is used to play the role of the user-agent in the OAuth flow. The native browser is a separate app and is not embedded in the client app, which provides the required isolation between the user-agent (native web browser) and the client app. The native web browser communicates with the client app through the provided mobile framework channels, where most mobile frameworks provide mechanisms that enable the communication and binding between different apps. For example, the Android architecture provides an intent messaging system for run-time binding between components in the same or different apps. The intent holds a description of the operation to be performed, and can contain data to be delivered to the receiver of the intent. The apps should inform the Android system (Intent Manager) about the intents they are willing to receive by registering intent filters. Each intent filter describes the intent of interest, and is associated with a component to respond to the intent. In early system native browser approaches the user was required to manually copy the access token from the authorization page and paste it in the client application, which was not convenient and was very confusing to users.

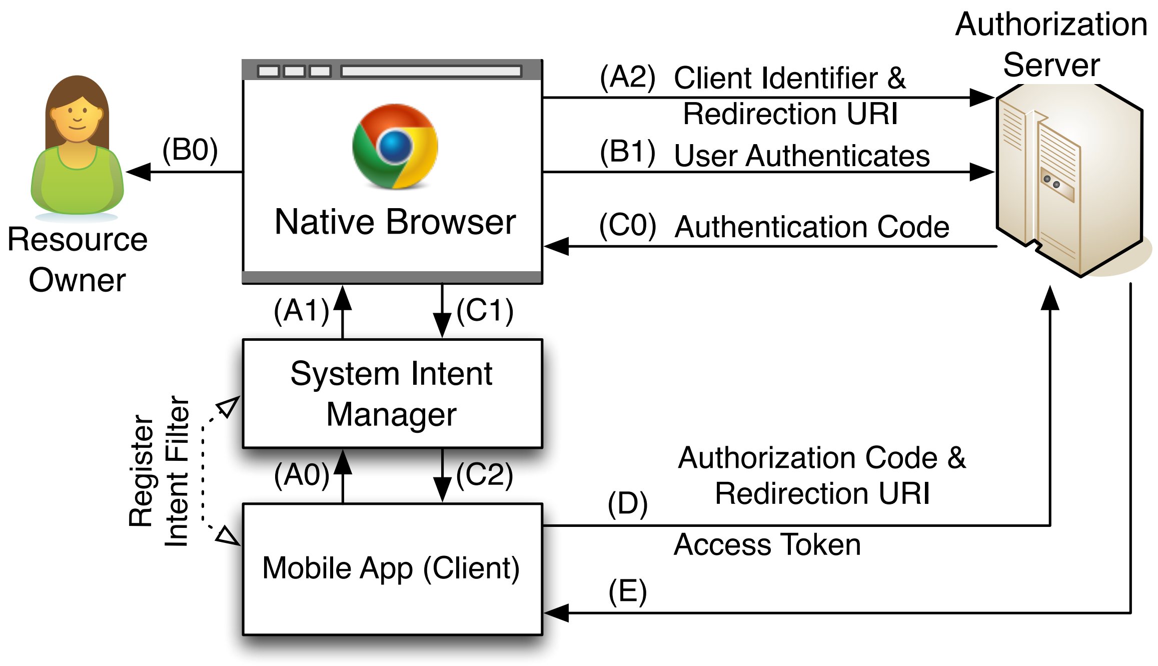


Figure 3: OAuth native system browser

The OAuth flow starts when the client app sends an intent `(view, http://auth-page, parameters)` to the system intent manager to view the authorization page, see Step A0 in Figure 3. The parameters sent in the intent include url, app id, callback url, and requested access (scope). The intent manager launches the native browser which loads the requested url (Step A1). The user is authenticated, the authorization server identifies the app and displays the authorization page to the user (Step B).

If the user agrees to grant the requested permissions, the authorization server redirects the native browser to authorization success page (callback url), which includes the authorization code in the url parameters (C0). To be able to pass the authorization code from the native browser to the client application, the callback url is set to a specific Multipurpose Internet Mail Extensions (MIME) which instantiates an intent to view this special callback url, for example the loading the url `db-key://connect?oauth_token=xx`, where the special MIME is `db-key`. To be able to receive intents for this special MIME, the client application should register an intent filter of the form`(view, db-key://.)`. Below is the required intent filter to be included in the client application manifest to respond to the special view request.

```
<intent-filter>
  <action android:name="android.intent.action.VIEW"/>
  <category android:name="android.intent.category.DEFAULT"/>
  <category android:name="android.intent.category.BROWSABLE"/>
  <data android:scheme="db-INSERT_APP_KEY"></data>
</intent-filter>
```

The intent manager receives the view intent from the system browser and locates the client application that registered an intent filter to receive this intent (Step C1 and C2). When the client app receives the intent it retrieves the access token embedded in the intent's data. The process is seamless to the user and client app does not communicate directly with the browser, instead all the communication is through the system intent manager.

2.2.4 Resource Provider Native App

This OAuth flow requires an installed resource provider native app as part of the authentication and authorization flow. For example, using the Facebook application as part of the authorization of a client app. In this approach the authentication and authorization processes are performed in the context of the installed resource provider app. This approach is similar to the OAuth flow using the native web browser, while the only difference is that the flow is processed through the installed resource provider app. When the user taps to start the OAuth flow, the client app sends an intent to the intent manager requesting to start the resource provider native app, and passes data parameters that include the client app id, secret, requested permissions (scope), and some other parameters. In some implementations the client app checks if the resource provider app is already installed, and if the app is not installed the user is asked and forwarded to the app market to install it. After the resource provider app retrieves the authentication code it sends it through the system intent manager to the client app, which then can use the auth code to complete the OAuth flow and retrieve the auth token. Figure 4, show the OAuth flow for this approach. This approach provides the required separation between the client and the user-agent (resource provider app) and some provider apps maintain a user authentication session, which does not require the user to re-authenticate during the OAuth flow and reduces the user effort. However, this approach requires the user to install the resource provider app for each client that requires to access a specific resource provider. In addition, each of the resource provider apps implements different authorization interfaces that might differ from their web interfaces, which can add to the factors affecting user perception of the security warnings and can lead to users disregarding the authorization prompts [55].

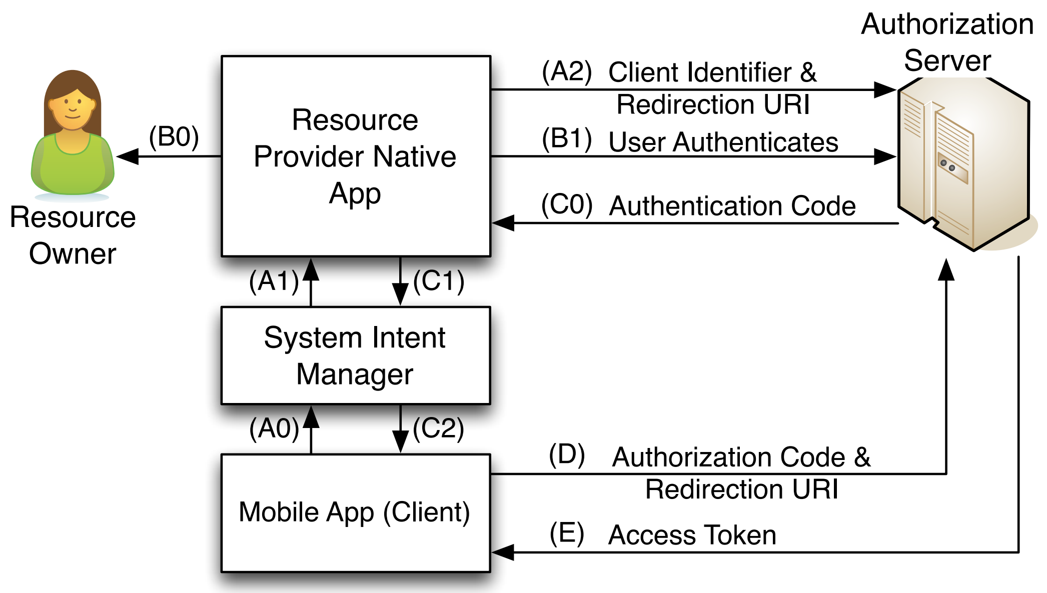


Figure 4: OAuth using resource provider app

2.3 Third-party Keyboards

Android was the first mobile operating system to allow users to install third-party on-screen keyboards (from here on out, simply referred to as "keyboards"). To be more precise, Android started supporting the installation of third-party keyboards since version 1.5 (known by its code name, Cupcake [4]). Users of other mobile operating systems, particularly iOS [77], have had a long standing criticism of Apple for not allowing the installation of third-party on-screen keyboards. However, with the introduction of iOS 8–the biggest iOS release ever according to Apple– third-party keyboards were finally allowed on both iPhone and iPad. However, if you look at the two platforms with regard to verifying third-party keyboards, you will find that iOS poses more limitations on keyboards than Android. Third-party keyboards can be seen as an opportunity to enjoy new features, but security experts, on the other hand, see them as a potential threat to users' privacy. The moment a user installs and activates a third-party keyboard on his/her mobile device, all of his/her keystrokes could be leaked or misused. This opens the door for malicious developers to create keyloggers for the purpose of spying and/or phishing for users' sensitive data. Furthermore, users may unknowingly install keyloggers from the online markets or may use a keylogger that a malicious user with physical access has installed on their devices. A malicious developer may build a keylogger from scratch or use an existing keyboard. In both scenarios, certain capabilities (permissions) must be available to these keyboards, such as the Internet capability. Thus, it is the responsibility of mobile operating systems, third-party keyboard providers, and smartphone users to ensure safety from these keyboards.

2.3.1 Input Method Editor

An IME is an Android application that contains a special IME service, a class that extends the *InputMethodService*. The service is required to be declared inside the application's manifest file. In addition, the manifest file must contain requests to the necessary permissions(the *BIND_INPUT_METHOD* permission is required to permit the service to connect the IME to the system), an intent filter that matches the *action.view.InputMethod* action, and other tags to specify the IME's characteristics. Moreover, the IME must have an input user interface or layout, for example, a layout that has the alphabet keys. Now, once activated and a user tries to enter data into an application's input fields, the layout is displayed and the user can press any key on that layout. The IME needs then to send the text to the application by either sending individual key events or editing the input around the cursor in the application‘s input field. In either way, an instance of the *InputConnection* class must be used to send the text. Users can install multiple IMEs on their devices but only one IME can be enabled at a time.

2.3.2 Third-party IME Risks

In this section, we describe the potential privacy risks concerning the use of third-party IME. We also explain what constitute risky IME and vulnerable device. Moreover, we provide some scenarios that might lead into a successful keylogging attack.

2.3.3 The Privacy Risks of Risky IME

A malicious third-party keyboard, once activated, can leak any data the end user enters into his or her device. The data can be text messages, emails, password credentials, phone numbers, social security numbers, credit card numbers, and more. Any of these data types can be misused to achieve damage, profit or both [16]. For example, an attacker may use stolen text messages to blackmail their senders or use stolen social security numbers to commit fraud. Thus, the severity of any keylogger attack depends largely on the device vulnerability level and user usage behavior. For instance, a keylogger attack against a rooted Android device that is used by an end user to do banking and economical activities may result in a severe damage or loss. For this reason, our user study is focused towards capturing and collecting users usage behaviors and possible device security misconfigurations.

There are many ways to build and distribute malicious third-party keyboards in Android. One such way is to convert a benign keyboard into a keylogger trojan and then make it available on non-official app stores. The conversion can be done by changing the Java[45] source code or Smali[34] source code and then compile and re-sign the Android application package (APK). Android apps are built using Java which is compiled to byte code to run on the Dalvik virtual machine[6]. Nowadays, many tools exist to reverse engineer Android APKs, modify their source code, compile them, and sign them back: *apktool* [70], *keytool* [61], and *APK multitool* [24]. In fact, an Android developer has used these tools before to demonstrate the feasibility of converting the *SwiftKey* keyboard into a keylogger that sends all keylogs to a particular server [15]. Alternatively, one could build a malicious keyboard from scratch. Android's Developers guide provides tutorials on building third-party keyboards [38]. Moreover, Android provides code samples for developers to download and use in their projects; for instance, an Android code sample for SoftKeyboard can be found here[39]. The malicious keyboards that are developed using the latter two methods can be then placed into both official and non-official online app stores. A malicious third-party keyboard attempting to steal user's entered would either store it on the device at first then send it to a remote server, or send it directly to a remote server [54, 13].

CHAPTER 3: THE PROBLEM OF IMPLEMENTING OAUTH IN MOBILE APPLICATIONS

In this chapter, the problem of implementing OAuth in mobile applications is discussed in more details. Starting with the root of the problem,

Platform	Resource Provider SDK	Embedded Web Component	Native Browser	Installable App	OS Integrated
Android	Facebook [19]	√		√	
	Twitter [60]	√			
	Dropbox [18]		√	√	
	Microsfot Live [52]	√			
	Box [8]	√			
	Google Plus [30]				√
	Instagram [42]	√			
	Linkedin [47]		√		
	Flickr [58]		√		
iOS	Facebook [20]		√	√	√
	Twitter [71]				√
	Dropbox [18]		√	√	
	Microsoft Live [53]	√			
	Box [9]	√			
	Google Plus [31]		√	√	
	Instagram [41]	√	√		
	Linkedin [59]	√			
	Flickr [57]		√		

Table 1: OAuth SDKs and authentication methods

3.1 Attacks Against OAuth Mobile App Implementations

In the introduction section we showed three implementations approaches for the OAuth in mobile applications: providers' Native App, Native Browser, and WebView-based. In this section, possible attacks against the former two are identified and discussed.

3.1.1 Attacks against System Native Browser

Compared to the embedded web component, the system browser provides stronger isolation guarantees. However, a malicious app can exploit the channel between the intent manager and the client app. For example, a malicious app could register the same intent filter as the client app, which could result in passing the the access token to the malicious app. If the user has installed both the client app and the malicious app, then the Android or iOS system will present the user with all the apps that registered to receive the intent and will ask the user to pick the app that should be used to receive the intent. The user can easily be tricked into selecting the malicious app. To demonstrate this attack, we downloaded an application from the Google Play Market, the application name is *GT-Document for Dropbox.* The app uses the *Dropbox SDK*, which uses the native system browser for the OAuth authorization and authentication steps. We analyzed the application's code and retrieved the application's intent filter from the application's manifest file. Then we created a malicious app called *GT - Document for Dropbox*, and we included the same intent filter retrieved from the legitimate app. The malicious app used the same app icon and name as the legitimate app, the only difference was the spaces included around the malicious app name. Now, if the user starts the the legitimate app *GT-Document for Dropbox*, authenticates, and authorizes the application, the authorization server sends the authorization code or token encoded in a URL. The system will prompt the user to choose between the two applications, Figure 5 shows the system alert. If the user chooses the malicious application, *GT - Document for Dropbox* can steal the access token and impersonate the legitimate app on the resource server.

3.1.2 Attacks Against OAuth Embedded Web Browser

Using an embedded web browser as a user-agent in a mobile app, enables the hosting client app to fully control the hosted WebView, which exposes the user-agent and does not provide the required isolation between the client and the user-agent. A

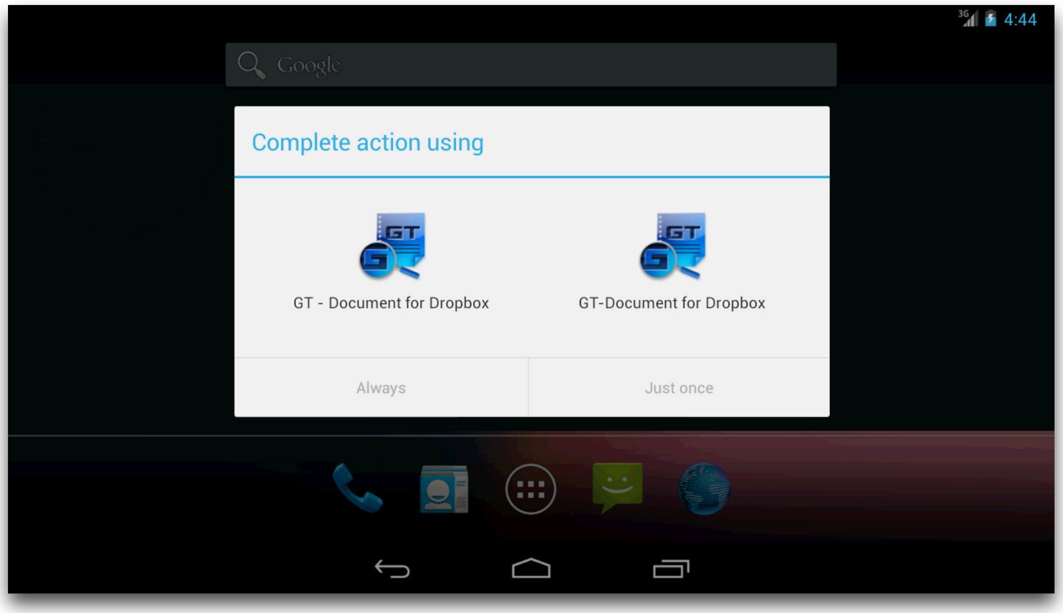


Figure 5: Attack using the system native browser

malicious hosting app can take control of the hosted WebView and launch attacks on both the user authentication and application authorization. In what follows, we demonstrate both attacks.

Stealing User Credentials. As discussed earlier the hosting app and the embedded WebView can communicate seamlessly. An attack to steal the user credentials (username and password) during the authentication stage can easily be executed by a malicious app in two stages, first injecting JavaScript in the WebView to retrieve the user's email and password upon clicking the submit button, and second registering a JavaScript interface that enables the embedded JavaScript to send the retrieved email and password to the hosting app. Below is the sketch of native and JS code:

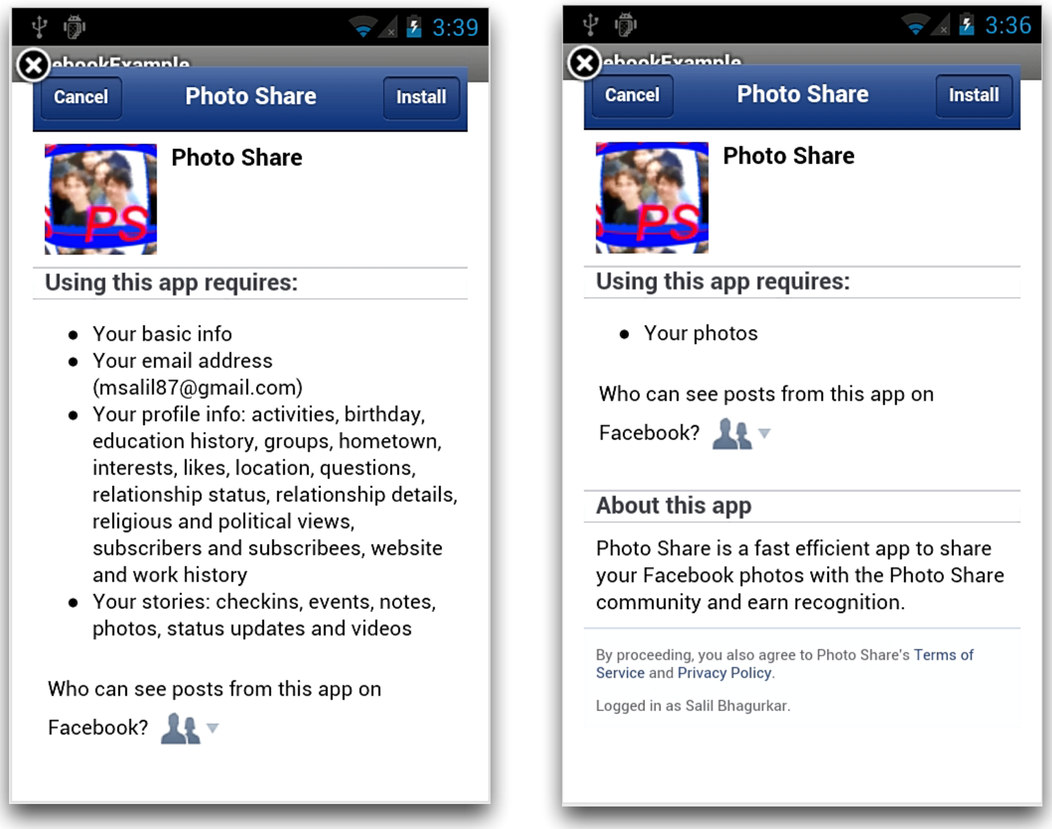
```
//Java (Native Client App)
myWebView.getSettings().setJavaScriptEnabled(true);
myWebView.addJavaScriptInterface(this , "JSInterface");
myWebView.loadUrl("javascript:" + contents of attack.js);
```

```
//JavaScript (attack.js)
var submitBtn = document.getElementById('btn_id');
submitBtn.onclick = function(){
  var email = document.getElementById('email_id').value;
  var password = document.getElementById('pwd_id').value;
  JSInterface.jsCall(email, password);
  return true;
}
```

The native app (Java) initializes the required JS interface, then injects the required attack JS. The embedded JS executes in the WebView and registers the required listener to capture the email and password inputed by the user. The retrieved information is sent to the native app via the `JSInterface.jsCall()` call. We developed this attack and tested it successfully on all the WebView SDK based OAuth implementations. The code presented is simplified and the actual code requires using class and type HTML selectors.

Modifying the Authorization Interface. Step B in Figure 2, the WebView presents to the user the authorization page which displays the list of permissions requested by the app being installed. Figure 19(a) shows our photo sharing Facebook application requesting access to the users' protected resources such as email address, profile attributes, checkins, events, notes, photos and many other permissions. Since the authorization page is displayed in a WebView, a malicious client app can easily modify the list of requested permissions displayed in the authorization page. This attack will trick the user into incorrectly interpreting the level of access requested by the app. Below is the JS code that should be injected into the WebView to replace all the requested permissions with simply "Your photos":

```
var permsUL = document.getElementById('perm_ul');
```



Cancel Photo Share Install

Photo Share

Using this app requires:

- Your basic info
- Your email address (msalil87@gmail.com)
- Your profile info: activities, birthday, education history, groups, hometown, interests, likes, location, questions, relationship status, relationship details, religious and political views, subscribers and subscribees, website and work history
- Your stories: checkins, events, notes, photos, status updates and videos

Who can see posts from this app on Facebook?

(a) Authorization Page

Cancel Photo Share Install

Photo Share

Using this app requires:

- Your photos

Who can see posts from this app on Facebook?

About this app

Photo Share is a fast efficient app to share your Facebook photos with the Photo Share community and earn recognition.

By proceeding, you also agree to Photo Share's Terms of Service and Privacy Policy.

Logged in as Salil Bhagurkar.

(b) Manipulated Page

Figure 6: Manipulating the authorization page

```
var permsUL.innerHTML = ‘<li><div>Your photos</div></li>’;
```

Figure 19(b), shows the manipulated authorization page. Note that the user authorizing the app in Figure 19(b) is tricked into thinking that the app is authorized to only access her photos, while the app is being authorized for all the permissions listed in Figure 19(a).

3.2 SDKs and Apps Analysis

We conducted an empirical study on the current OAuth implementation trends followed by the service providers and by the OAuth development choices made by application developers. We reviewed the OAuth based SDKs supported by the popular resource providers such as Facebook, Twitter, Dropbox, Microsoft Live, Google Plus, Box, Instagram, LinkedIn, and Flickr. Table 1 shows the the types of OAuth implementations adopted by the SDKs of the different resource providers for both the Android and iOS frameworks. As can be noted from our results, some SDKs include more than one OAuth implementation. For example, the Dropbox SDK for Android provides both native browser and installable provider app based OAuth implementations. It is also notable, that several SDKs such as Microsoft Live, Box and Instagram for Android only support WebView based OAuth flows. Also for iOS, the Microsoft Live, Box and LinkedIn only support UIWebView based OAuth flows. This shows that providers have varying security aware implementations of the their OAuth SDKs, where some providers are more security aware than others by limiting their implementation to more secure OAuth flows such as native browsers or installable apps.

To study the behavior of SDKs that provide multiple OAuth flow implementations we focus on the Facebook SDK for Android, which allows the OAuth flow to take place either through a WebView or through the Facebook installable app. The default Facebook Android SDK [19] initially queries the system to check if its Facebook app is installed, if it is already installed the OAuth flow will proceed through the installed Facebook app, otherwise an embedded WebView will be used. The SDK is open source and app developers can adapt its behavior. In order to study how the developers adapt the default Facebook SDK, we downloaded from Google Play Android market popular applications that integrated with Facebook and Dropbox services. The download application packages (APK) were used to extract the .jar files using the `dex2jar` tools, then the source files were extracted from the jar files by using the `jd-gui` decompiler tools. For each app, we analyzed the source code and reviewed the OAuth flow adopted. We analyzed 231 Facebook integrated apps, 68% of these apps imported the Facebook SDK and 32% did not use the Facebook SDK and included their own implementations of OAuth. For the apps the used the Facebook SDK we found that 56% of the developers imported the SDK without any changes, 22% of the apps chose to force the WebView based flow (OAuth Dialog), and 22% requested authentication using the installed Facebook App. On the other hand, for the apps that did not use the Facebook SDK we found that 68% used the WebView based flow, only 3% requested authentication using the installed Facebook App, and 29% implemented a flow similar to the default Facebook SDK flow. Figure 7, summarize the statistics collected for the Facebook integrated apps. We also found that, only 3% of all apps prompted the users to install the Facebook app from the market upon the first run.

We analyzed 202 Dropbox integrated apps, 50% of these apps imported the Dropbox SDK and the other 50% did not use the Dropbox SDK and included their own implementations of OAuth. All the apps the used the Dropbox SDK did not change its default behavior and adopted both the native browser and Dropbox app flows. On the other hand, for apps that did not use the Dropbox SDK we found that 26% used the native browser flow, and 74% implemented a flow similar to the default Dropbox SDK flow. Figure 8, summarize the statistics collected for the Dropbox integrated apps. We also found that, only 0.5% of all apps prompted the users to install the

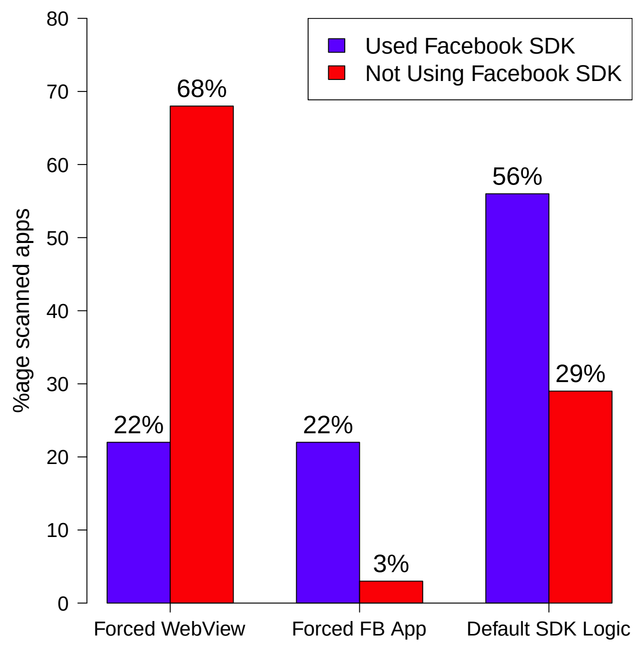


Figure 7: Facebook OAuth implementations

Dropbox app from the market upon the first run.

3.3 Threat Model

The dissertation's threat model is a JavaScript injection into the WebView in order to achieve any of the following:

Obtaining Users' Credentials. When the mobile app includes the WebView component in its source code, it gains full control. For instance, the mobile app can communicate seamlessly with the WebView before, during and after the OAuth flow. This capability gives the mobile app the power to intercept any URL loading into the WebView; hence, it can change it or modify it. Therefore, one possible attack is to obtain the user's credentials during the authentication phase. This can be achieved by first injecting JavaScript in the WebView to retrieve the user's email and password upon clicking the submit button and second by registering a JavaScript interface that enables the injected JavaScript to pass the credentials to the mobile app. The cre-

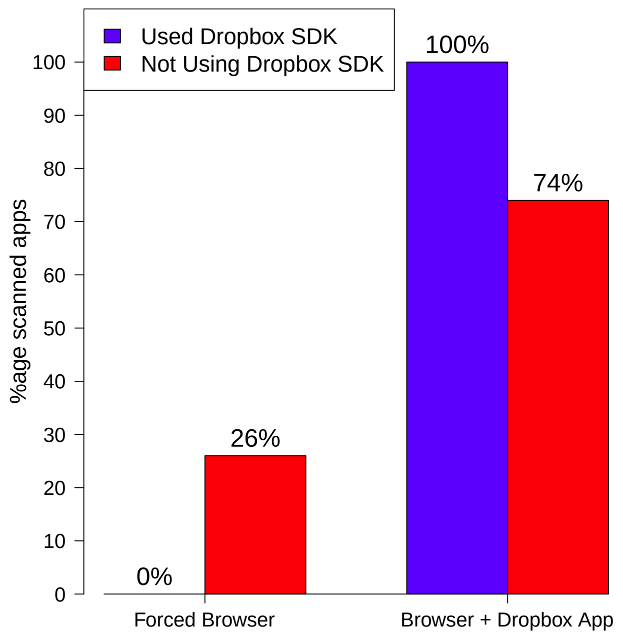


Figure 8: Dropbox OAuth implementations

dentials then can be sent to a remote server decided by the malicious developer.

Manipulating the Authorization Page. During the OAuth flow, the WebView presents the authorization page to the user, which shows the list of permissions requested by the mobile app. The permissions list plays a critical role on the user's decision whether or not to allow access. A malicious developer via her/his mobile app can trick the user into incorrectly interpreting the level of access requested by the app. This can be acheived by modifying the list of requested permissions; for instance, removing the dangerous permissions from the list.

In the evaluation in section 4.3.7, we show an example of manipulating the Facebook authorization page.

3.4 Adversary Model

In this paper, we assume a malicious mobile app developer bases his/her attack on exploiting the OAuth-WebView vulnerability. The malicious developer implants the attack into a functional application and then uploads it into an online app market. The malicious app functionality is centralized in connecting mobile users to their profiles on common web sites (service providers). The victim downloads the app and installs it on his/her mobile device. At the first run, the victim has to go though the OAuth flow, which entails submitting his/her credentials. The malicious app can either manipulate the permissions list and/or obtain the credentials. For the impersonation attack, the malicious developer can register his or her malicious app to listen to the same specific data request that the client app (benign) is registered to listen to, which could result in passing the access token to the malicious app.

3.5 Related Work

To begin with, the past six years have witnessed the discovery of numerous security flaws and feasible attacks against OAuth [36, 73, 75, 26, 25, 35, 37]. To narrow down our discussion, we split our related work section into three parts: OAuth related part, WebView related part, and OAuth-WebView related part.

OAuth:OAuth client-flow has been a source of several security concerns to the research community. For instance, the work of Sun and Beznosov [69] made a strong recommendation about not putting the OAuth protocol at the hand of developers due to the fact that client-flow is inherently insecure. R. Paul [64] emphasized an existing security flaws in Twitter OAuth implementations for desktop applications that risk client‘s credentials. Pai et al. [67] performed formal verification to the OAuth protocol and found additional security flows. The majority of service providers nowadays provide developers with software development kits (SDKs) to include them in mobile applications to seamlessly integrate them with their services. Wang et al. [65]

formally verified some of these SDKs and the apps that are built by importing them. The apps were found vulnerable to serious exploits that risk OAuth secrets, which are, session IDs, app secrets, codes, access_tokens, and refresh tokens.

WebView: The previous researches on WebView vulnerabilities were conducted in isolation to the OAuth protocol. For instance, Luo et al. [48] explained the risk of using WebView and its APIs on web's security. Their contributions were in discovering different kind of attacks and analyzing their fundamental causes. These works have no suggested solutions to any of the discussed attacks. Wang et al. [74] has conducted the first systematic study on unauthorized origin-crossing over mobile operating systems, including Android and iOS. Their study pointed out the WebView as a mean to lunch the unauthorized origin attacks. The focus of this work is on protecting benign apps, while our focus is on differen kind of attacks in malicious apps. Luo et al. [49] explored the "touchjacking" attack on different platforms, including Android, iOS, and Windows. The attack targets the weaknesses of WebView pertaining the handling of touch events. Our focus is on WebView's javascript capability.

OAuth-WebView:A recent work [10] pointed out that migrating OAuth into the mobile arena, without careful measures, had resulted in that 59.7% of mobile-capable applications are vulnerable to several kind of attacks. The authors explained the risks of using WebView in delivering the OAuth tokens. Hunt [50] has recommended that developers and end-users must be instructed to trust only external System-Browsers. Their recommendation is based on the fact that embedded browsers could be misused during the end-user authorization process to display its own user-interface instead of allowing trusted browser to render the authorization user interface. This could result in that all information that are exchanged during the authorization step to be logged and users would not notice. A. Wulf [3] showed how mobile applications developers can still obtain users' credentials despite the usage of OAuth protocol.

Others: Advertising in Android applications have proven to cause many security and privacy problems that led into leaking users' private information. Felt. et al. and Shekhar et al. in their works [62, 68] proposed frameworks based on the privilege separation principle. Both speculated that their ideal deployments are as part of the core Android distribution, similar to our approach with SecureOAuth.

CHAPTER 4: PROPOSED SOLUTIONS

In this chapter, a set of potential and proposed solutions for hardening the OAuth-WebView implementations in mobile applications are discussed. Starting with the potential solutions then moving into the proposed ones.

4.1 Potential Solutions

We first start by discussing some potential solutions to the problem of JavaScript injection attacks against OAuth pages.

4.1.1 Byte-code Re-Writing

A second approach would be to use API-level access control using byte-code rewriting, which is to provide modified behaviors that allow fine-grained access control at the Java API level (WebView). The modified behaviors are achieved by implementing secure wrappers (SecureWebView) that replace the original Android WebView APIs. Therefore, before invoking any of these APIs, defined access policies are enforced (e.g. no javascript injection is allowed when loading OAuth URLs). Applying this approach requires three main steps. First, the bytecode is extracted from the APK and decompiled, then the occurrence of the security sensitive APIs in the code is identified using statical analysis tools. Second, the behavior of these occurrences is then modified to include the added access control logic. Finally, the bytecode and the modifications are repackaged into a single APK file using a new digital signature. The drawback of this approach is that it assumes that applications either do not possess native code or their native code is deprived from being executed. Thus, this approach is not complete and does not cover the cases where attackers use native code to call the native library; thus, bypassing the defined access policies.

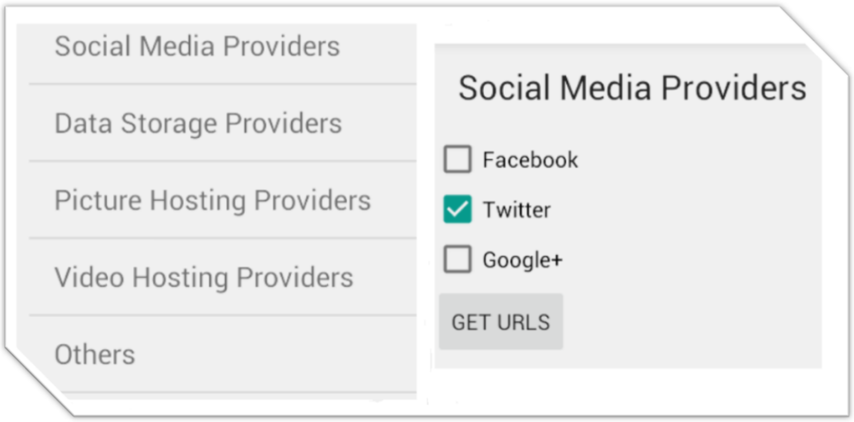


Figure 9: The `SecureOAuth` system app is used to add common service provider URLs into the `ProtectedURLsDB`.

4.1.2 Moving Target Defense

Moving target defense can be used to randomize the names of the sensitive HTML elements that are part of the OAuth authentication and authorization pages. The success of the JavaScript attacks against both pages depend largely on knowing the structure of these pages and the names of the HTML elements that contain the credentials and the permissions information. The randomization can be done per user's session per request to prevent attackers from learning the changes plan. The moving target defense can be applied on the web server side instead of the mobile/client side. The technique can only work to defend against the attacks that target the authorization page but not the authentication page because attackers can launch a more aggressive attack by registering event handler on all the input fields of the authentication page and copy all entered data.

4.2 Proposed Solution 1 : Privilege Separation - OAuthManager

We propose to use the privilege separation [63] concept to ensure that the client application has no control over the user-agent. We removed the critical OAuth components and implemented it in a separate application (secure sandbox), which we refer to as the *OAuth Manager*. While many aspects of the proposed solution is applicable for other smart phone platforms (iOS, and Windows Phone), we focus our discussion on Android platform.

4.2.1 Solution Details

The user-agent used is a WebView embedded in the trusted *OAuth Manager* app, which isolates it from the client application and can be accessed only through the secured channel that is managed by the system (Intent Manager).

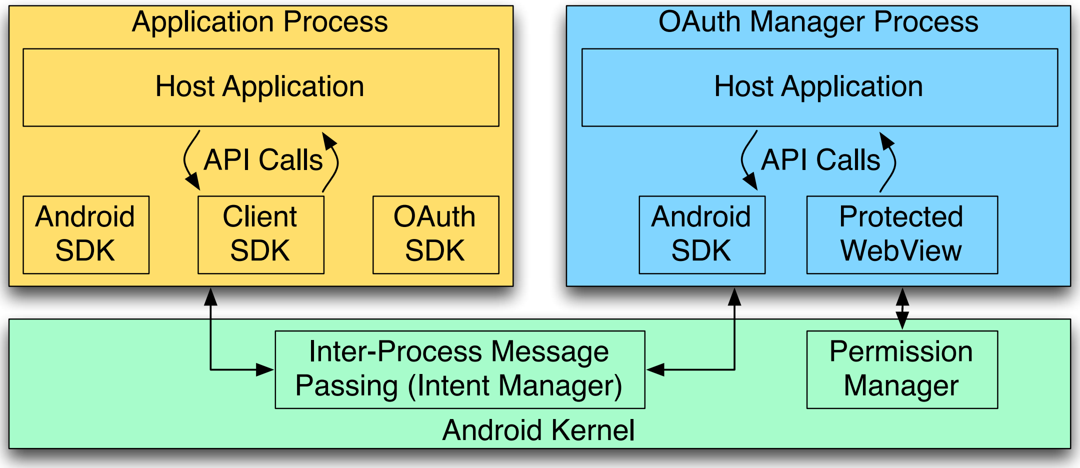


Figure 10: OAuth Manager running on a separate application and communicating with the client apps via the intents

Figure 10, shows the *OAuth Manager* framework. The OAuth Manager is implemented as a separate application, and is responsible for displaying the authorization and authentication in an embedded web component hosted in the OAuth manager and is accessible to the client app. The flow starts when the client application sends an intent to the Intent Manager requesting to start the OAuth Manager application and passes it the required OAuth parameters such as the client app id, secret, and requested permissions (scope). *OAuth Manager* is started and passed the required parameters. The *OAuth Manager* ensures that the client application is a signed application and contacts the systems permission manager to verify that the client application is granted the internet permission `android.permission.INTERNET`, which avoids the privilege escalation scenario. The OAuth flow would terminate if the client application is not granted the internet permission, otherwise the authentication and authorization steps are completed, and the auth code is retrieved in the web component embedded in the *OAuth Manager*. This form of isolation will ensure that the client application is not able to manipulate and control the authentication and authorization pages. Moreover, for the purpose of preventing from the impersonation attack (hijaking the token) *OAuth Manager* includes a routine that works upon running it on the device to the first time and whenever a new app is installed. The routine is meant to verify that no other app is registered to handle intents similar to the one that *OAuth Manger* handles. In the manifest, we statically register a Broadcast receiver that listens to the following system actions: `android.intent.action.PACKAGE_INSTALL` and `android.intent.action.PACKAGE_ADDED`. The system will notify *OAuth Manager* if a new package is installed. In the OnReceive method, an instance of the PackageManager is used to filter the apps that included in their manifests an intent filter for the *OAuth Manger* custom intent. If such apps exist, *OAuth Manger* notifies the user of their existence and ask her to uninstall them. As the final step, the *OAuth Manager* sends the retrieved OAuth access code to the client application through the

Intent Manager as a result to the requested intent result. Then the *OAuth Manager* automatically finishes and destroys its process. To the user the execution is transparent and is very similar to the WebView experience. To the developer the SDK can be easily updated to open the *OAuth Manager* instead of opening an embedded WebView, which is a very minor change to the original resource provider SDK.

4.2.2 Performance

For the purpose of comparing our proposed *OAuth Manager* approach with the other OAuth flow implementations, we conducted a performance study based on memory consumption and response time. We performed our experiment on a standard Android developer phone, the Nexus S, that has android version 4.1.2, 1007.89 MB internal memory, 13624.34 MB SDCard, 343 MB RAM, system browser 4.1.2-485486. The performance analysis is focused on studying the overhead caused by adopting our method as an authentication and authorization method.

4.2.3 Response time

We performed benchmarking to estimate the overhead of OAuth manager on displaying the authentication page. It is the required by the OAuth flow implementation to complete the loading of the authentication page after the user authentications. For this purpose, we used Android Logging System, we added hooks to the code to record the time samples immediately after the user clicks the login button, and promptly after successfully loading the authentication page. Our experiment is conducted using the Facebook OAuth flow. Table 2 shows the time in milliseconds required by the different OAuth flow implementations. The *OAuth Manager* is faster than the system browser and the embedded WebView. The *OAuth Manager* is slightly slower than the Facebook App, this is because the Facebook App does not use any embedded WebViews and relies on simple api calls. In addition, it is important to note our code was not optimized and was designed to provide OAuth flows for different service provider apps.

Method	Response(milliseconds)
System Browser	3429
Embedded WebView	8077
Facebook App	1879
OAuth Manager	1892

Table 2: Comparison of response time (milliseconds)

4.2.4 Memory overhead

We used the Android Debug Bridge (adb) to measure memory overhead. The adb is a versatile command line tool that enables us to communicate with an emulator instance or a connected Android-powered device. In our case, we used Android-powered device, Nexus S. We ran our test application multiple times and each time we used different a authentication method. We recorded the memory consumption for each method. In interpreting our results we are primarily concerned with the proportional set size, which is (Pss) the amount of memory shared with other processes, divided equally among the processes who share it. Table 3 shows that *OAuth Manager* memory requirements a memory footprint lower than the native browser and the Facebook App. However, the *OAuth Manager* requires more memory than the embedded WebView, this is due to the required security checks. In addition, the *OAuth Manager* code was not optimized for memory usage. The embedded WebView has shown to be insecure and has the possibility of leaking users' sensitive data. In contrast, *OAuth Manager* offers a secure solution and protects the content from being stolen or manipulated.

4.2.5 Security Analysis

The OAuth flow based on *OAuth Manager* is more secure than the other flows: 2.2.3 and 2.2.2. *OAuth Manager* flow is safe and provides the measures to prevent from the three attacks: the two in Section 3.1.2, and the attack in Section 3.1.1. By taking the WebView component out and hosting it in a separate isolated process, it removes the vulnerabilities that are caused by WebView based attacks. Moreover, it uses explicit intents plus a routine that is called upon installation and upon installing any new app to avoid other apps from listening to the same intent and thereby avoiding attacks discussed in Section 3.1.1.

Method	Memory (kB)
System Browser	41386
Embedded WebView	5525
Facebook App	22114
OAuth Manager	13518

Table 3: Comparison of memory consumption (kB)

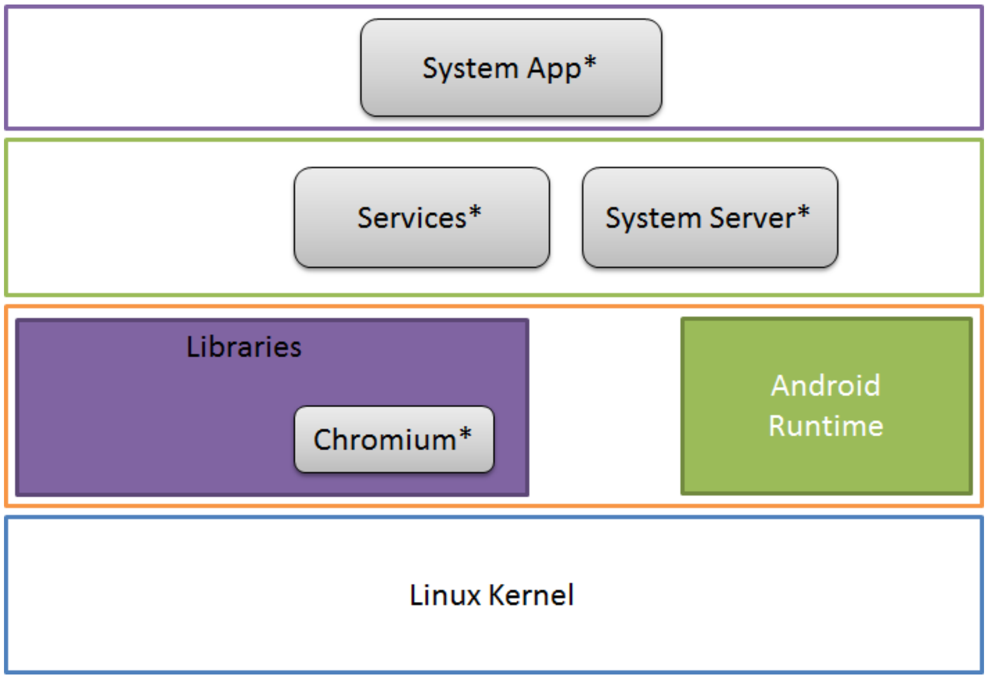
4.3 Proposed Solution 2: Refactoring the WebView - SecureOAuth

In this section, we talk about our second solution which is based on refactoring the WebView component.

4.3.1 Proposal

We propose to use a *whitelist* (or *white list*) access control mechanism to address the threats outlined in Section 3.3. In general, whitelist allows JavaScript injection in any page that is loaded in a WebView, unless that page has been included in the protected URL list. The solution blocks any JavaScript injection attempt against

Figure 11: Solution Architecture



these protected URLs in order to safeguard the privacy of the user's data. In this context, we suggest checking the loaded URL before allowing the JavaScript injection. The goal is to prevent JavaScript injections on particular URLs (OAuth authentication and authorization URLs) while being loaded in the WebView,thus protecting the user's privacy and security. So, we need a mechanism to enforce these checks, to store OAuth URLs, and to give users some control. Before we talk about our proposal, we first go over other solutions and discuss their feasibility.

4.3.2 Our Proposal: SecureOAuth

We propose *SecureOAuth* framework that hardens OAuth-WebView implementations in Android applications. In this solution, the OAuth-capable applications' APKs do not have to be decompiled and repackaged again. Instead, we implement the access control on the Android layers, including, application layer, framework layer, and library layer. SecureOAuth is composed of three parts, Figure 11:

- SecureOAuth, Android System Application: This application runs on Android application layer, and provides users with the ability to view the protected OAuth URLs, delete them, and add new ones.
- SecureOAuthService, Android System service: This service runs as a thread in the *system_server* process. It is responsible for creating a database named *ProtectedUrlsDB.db* in the *data/system* path at device bootup, initializes the tables and assigns permissions to the database. The purpose of the permissions assignment is to allow only a system process or app like SecureOAuth to view, add or/and delete URLs to/from the ProtectedUrlsDB database (read/write permissions), while allowing user applications the ability to read from databse, but not to write to it (read permission).
- Modifications in the (Webkit/Chromium) Library: We modified the library to include new checks for filtering or blocking javascripts. The Chromium library is the rendering engine of the Android WebView that has filters for the javascript. The entry point for loading URLs in Chromium library/Webkit library is modified to check for any javascript injection in case any of the protected OAuth URLs–from the ProtectedUrlsDB database– is being loaded. If any javascript injection occurs , then it is blocked in the library level.

4.3.3 Implementation Details

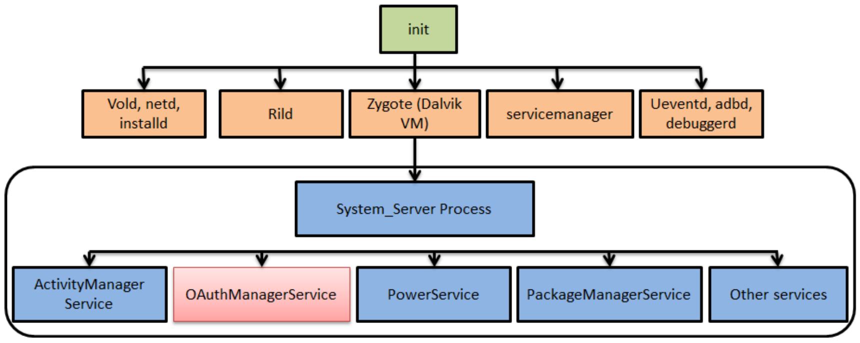
We implement our SecureOAuth proposal to illustrate its details and demonstrate its feasibility. SecureOAuth consists of three parts: a system application, a new

Android service, and library modifications. Our proof-of-concept implementation of the SecureOAuth prototype integrates these three components into the Android Open Source Project, version 4.4.2 (Kitkat). In this section, we show the detailed code modifications and steps taken to implement the SecureOAuth framework.

4.3.4 Chromium Library Modifications

Starting from Android 4.4 Kitkat, the WebView is rendered by the Chromium library instead of Webkit. The Chromium library c++ code is modified to include new checks for the URLs before loading them into the WebView (see Algorithm 0.1). Before loading a URL in the WebView, the library verifies whether it is a javascript injection or not. If it is not a javascript injection, the loading continues normally. Otherwise, the currently loaded URL in the WebView is verified using the URLs in the ProtectedUrlsDB database. If it matches any of them, the javascript injection is then blocked from executing; thus, protecting the loaded web page from being tampered with or it‘s data from being stolen. The modifications is applied to the *content_view_core_impl.cc* file (see Appendix C), the file contains the implementations for loading URLs and executing javascripts. The ProtectedUrlsDB database is assumed to exist at predefined path (data/system/ProtectedUrlsDB.db), the database is created by the service 4.3.6. Moreover, javascript blocking is done in both *LoadUrl* and *EvaluateJavascript* methods to allow backward compatibility across all versions of android (e.g. Applications whose target is below 4.3). Note that the URL verification step requires reading permission on the database. Because of that, the reading permission is granted to all user applications. The Chromium library runs the verification step in the context of the these applications; therefore, it posses the needed reading permission.

Figure 12: SecureOAuthService family tree.



4.3.5 SecureOAuth: System Application

The purpose of creating *SecureOAuth* system application is to provide users–curious users–some control over the protected URLs database. Using SecureOAuth application, user can view all URLs, remove URLs, and add new URLs; therefor, the app requires read and write permissions over the ProtectedUrlsDB database file, the permissions are assigned by the service 4.3.6. This app is packed as system app by assigning system id to it in the *AndroidManifest.xml* file as *android:sharedUserId=" android.uid.system"*, adding the platform certificate in its *Android.mk* file, *file.LOCAL_CERTIFICATE:=platform*, and packing it into the list of system apps by adding the app to the list in *generic_*
no_telephony.mk for the purpose of emulator compilation (see Appendix C).

Algorithm 0.1: CHECKURL(*url*)

```
if $isJavascript(url)$
  then {
    $currentURL = GetWebContents().GetURL()$
    if $isInProtectedUrlsDB(currentURL))$
      then { $BlockURL(url)$
      else { $continuenormalexecution$
  }
  else
  then { $continuenormalexecution$
```

4.3.6 SecureOAuthService: System Service

This service is responsible for creating the *ProtectedUrlsDB* database under the */data/system* path during bootup. After creating that database, the service sets the file permissions, creates the necessary tables, and populates them using an initial list of protected URLs. This service is made part of Android‘s system server by adding it into the *SystemServer.java* file [76]. The system server is the core of the Android system that starts as soon as Dalvik is initialized and running as illustrated in Figure 12. SecureOAuthService as well as other system services are running in the context of the system server process. SecureOAuthService is created from the Android Interface Definition file (.aidl) located at *frameworks/base/core/java/android/os/*. The current implementation of SecureOAuthService has no exposed functions that other processes or applications can access. However, for future extensions, this implementation can be extended to include such functions.

Listing 1: Lines added to SystemServer

```
try{
Slog.i(TAG, "SecureOAuth service);
ServiceManager.addService("OAuth", new
SecureOAuthService(context});
}catch{Throwable e){
Slog.e(TAG, "Failure to start secure
OAuth service", e);
}
```

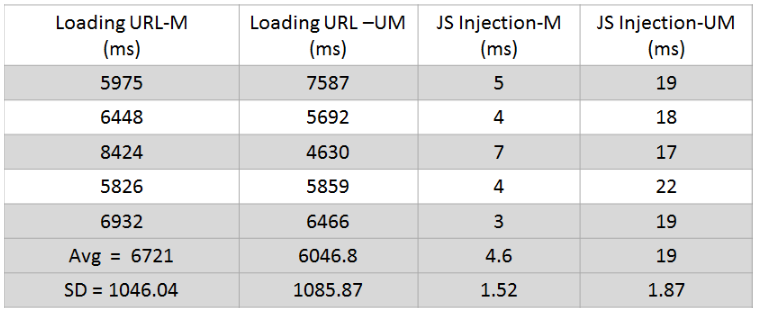
4.3.7 Evaluation

The evaluation study of our proposed solution, *SecureOAuth*, consists of three parts: we first measure the effect of our solution on loading URLs (JavaScript) in the WebView. Second, we measure the memory overhead of the *SecureOAuthService* and *SecureOAuth* app and lastly, the security analysis. Performance tests for URLs loading time and memory footprint for the system server process after our solution was implemented were done by adding hooks to the code before and after loading a URL. We performed our experiment on an Android emulator, Nexus 4, that has Android version 4.4.2, icd.density=320,$\tilde{2}$ GB RAM, disk data partition size=200M, heap size=64MB, and HVGA resolution. In the experiments, the URL `http://www.google.com/mobile/android` is loaded in a WebView, then a JavaScript code is injected. The script changes the color of the text "Google Apps for Android"– displayed in the page– from black to red.

We also used the OAuth URLs of five different service providers, mainly, the URLs of their authorization and authentication pages. For instance, Figure 15 shows the original authorization page of a Facebook app and a modified version–we used a script to manipulate the permissions' list in that page.

4.3.8 Processing and Memory Overhead

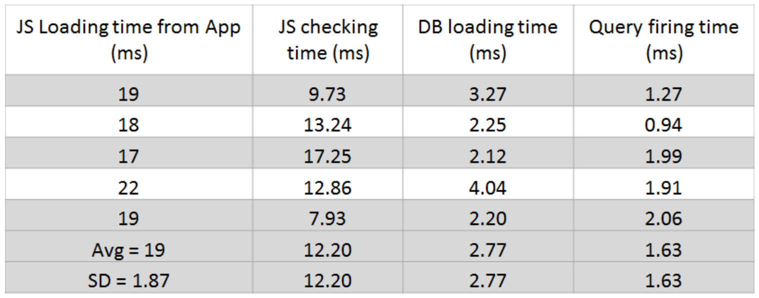
For the purpose of measuring the overhead caused by our solution, SecureOAuth, we conducted a performance study based on the response time and the required memory. These tests were carried out by comparing a fresh compiled system image without any changes running on standard QEMU emulator against a system image with SecureOAuth solution implemented running on standard emulator. The codebase used for both the tests were Android Kitkat 4.4.2 _r1.



Loading URL-M (ms)	Loading URL –UM (ms)	JS Injection-M (ms)	JS Injection-UM (ms)
5975	7587	5	19
6448	5692	4	18
8424	4630	7	17
5826	5859	4	22
6932	6466	3	19
Avg = 6721	6046.8	4.6	19
SD = 1046.04	1085.87	1.52	1.87

Figure 13: The difference in response time between a fresh image (UM) and a modified image (M) of the Android Kitkat version.

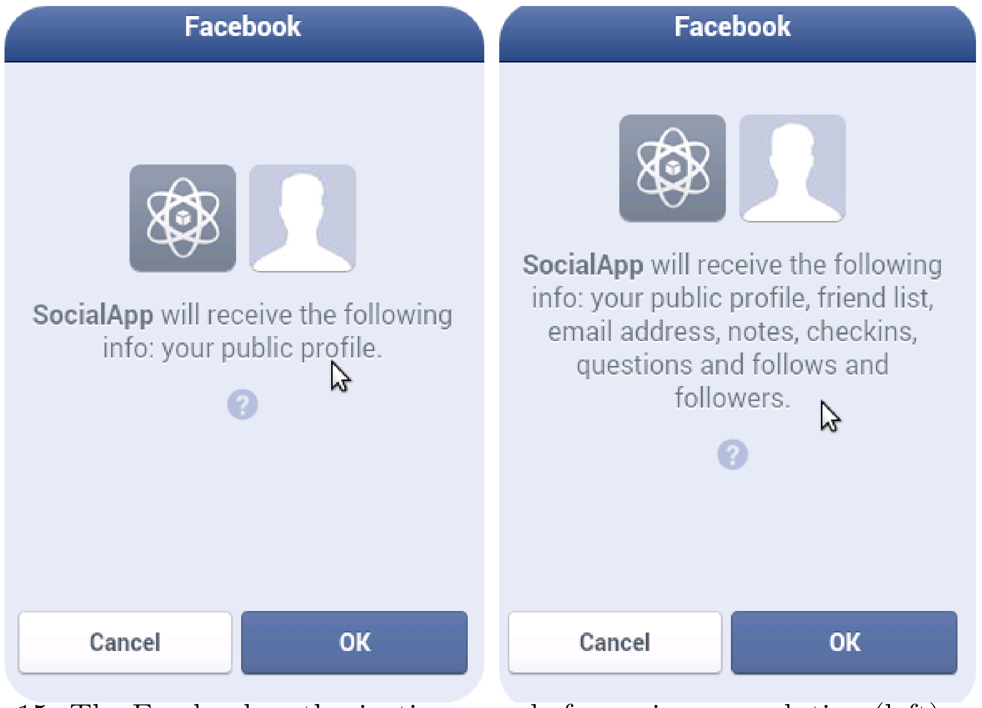
Response time: We performed benchmarking to estimate the overhead of the SecureOAuth framework in verifying the JavaScript injections before executing them. The Chromium library needs to review any JavaScript injection attempt before actually executing it. The review process must have some performance overhead; thus,



JS Loading time from App (ms)	JS checking time (ms)	DB loading time (ms)	Query firing time (ms)
19	9.73	3.27	1.27
18	13.24	2.25	0.94
17	17.25	2.12	1.99
22	12.86	4.04	1.91
19	7.93	2.20	2.06
Avg = 19	12.20	2.77	1.63
SD = 1.87	12.20	2.77	1.63

Figure 14: The performance overhead is being split between the different implementation's segments.

we performed some tests to measure such a delay. In so doing, we added some hooks into the Chromium source code in many places. We also added some hooks into an app that has a WebView component to measure the time before and after attempting a JavaScript injection. We first measured the difference in response time between a fresh and modified copy of the Android Kitkat version. In Figure 13, we show the results of our tests. There are slight differences measured in milliseconds between the two images. Figure 14 shows the split of the added overhead among the different implementation's segments, the first column shows the delay in milliseconds that the user experience whenever there is a JavaScript attempt that is measured from the application layer. The average delay time is 19 milliseconds. The second column shows the same delay but when measured from the library layer. The new Chromium's check is mainly composed of loading the ProtectedUrlsDB and firing a query. So, the third and fourth columns show time taken for loading the ProtectedURLsDB database and firing the query respectively. Due to the use of hashing and indexing the framework can scale up to support large number of URLs without affecting the performance. We conducted repeated performance tests with a varying number of URLs in the database and the results show no performance degradation.


Facebook

SocialApp will receive the following info: your public profile.

Cancel OK

Facebook

SocialApp will receive the following info: your public profile, friend list, email address, notes, checkins, questions and follows and followers.

Cancel OK

Figure 15: The Facebook authorization page before using our solution (left) and after (right).

Item	Memory(KB)
Image-Original	17789
Image-Modified	19464
SecureOAuth App	5296

Table 4: Memory Overhead.

Memory footprint: For the purpose of measuring the memory footprint of the system server for original image and modified image as well as the SecureOAuth app, we used the adb shell dumpsys meminfo command [5]. Table 4 shows the values that we obtained from running the command; the SecureOAuth service adds only 1676 KB memory overhead and 5296 KB for the SecureOAuth app. The solution has a bounded memory overhead.

4.3.9 Security Analysis

For the purpose of testing the effectiveness of our proposed solution in denying OAuth-WebView implementations related attacks in particular and JavaScript injec-

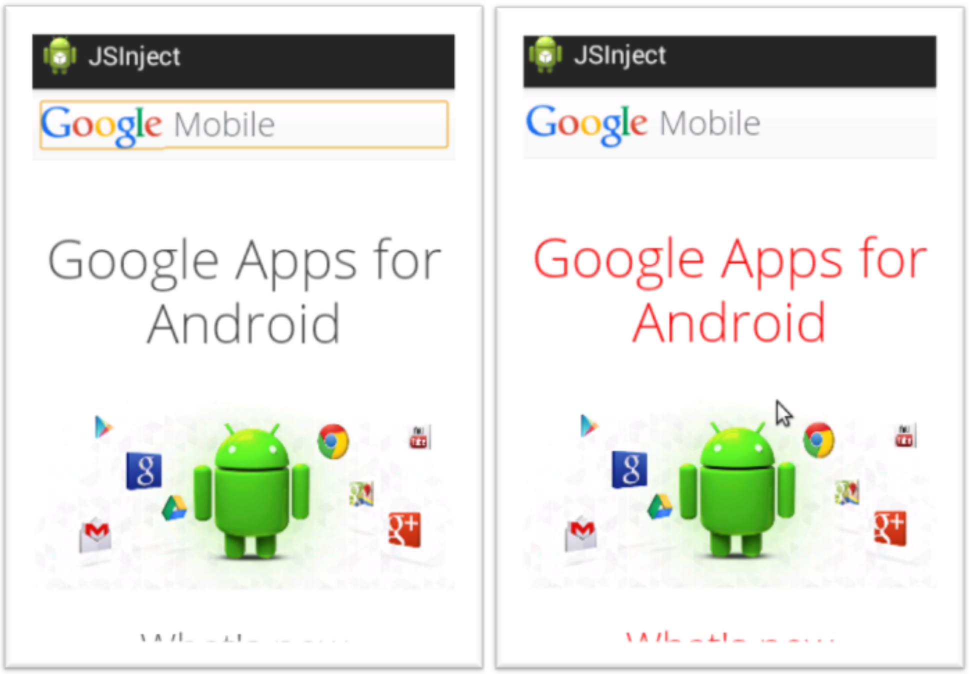


Figure 16: The page that we used in testing our solution, the page before js injection (left) and after js injection (right) .

Attacker Skill Level	Approach
Basic	Inject the malicious JavaScript into providers' WebView-based OAuth implementations
Advanced	Write his own WebView-based OAuth implementations to inject a malicious JavaScript
Expert	Write a native code that calls the native library (Chromium) to inject the malicious JavaScript

Table 5: The different attacker's skill levels that we considered for the WebView-based attacks.

tion in general. We populated the SecureOAuthURLsDB with the OAuth links of the five resources providers known to be using WebView in their SDKs (Table 1) in addition to the URL of the test page (Figure 16). Then, we asked a group of five undergraduate students with security and JavaScript advanced knowledge, to try to penetrate the pages, which means to try to achieve any of the following: steal users' credentials from the authentication pages, modifying the permissions list in the authorization pages, and changing the font color of the test page. The students first used the Chrome DevTools to debug the web pages live on an Android device [32] in order to understand the structure of these pages and locate the target HTML elements. The students individually and then as one group and collectively generated fifteen different attacks–malicious scripts that are injected into the WebView after loading the target pages. All attacks were successfully denied and loading the pages continued normally. As such, *SecureOAuth* framework handers OAuth-WebView implementations in smartphones by applying the whitelist access control mechanism on the library layer supported with a system service and a system application. The hardening assumes the threat and adversary models, defined in Chapter 4. Compared with the two approaches discussed in previous chapter, SecureOAuth is more secure as it assumes more sophisticated attacks than the two as explained previously in Table 5.

Summary: The evaluation study reports that SecureOAuth does not require high memory overhead. Moreover, the solution adds an acceptable processing overhead in case there is a JavaScript attempt–given that the SecureOAuth code is not yet fully optimized for memory and performance usage. Moreover, SecureOAuth secures OAuth-WebView implementations effectively. Table 6 compares between SecureOAuth framework and the other two solutions discussed in previous chapters. Thus, it is recommended to adopt SecureOAuth for solving the OAuth security flaws that were discussed in this work and could also be extended to solve other WebView security flaws.

Factor	**OAuth Manager**	**Byte-code Rewriting**	**SecureOAuth**
Overhead	Low	High	Moderate
Assumption	Basic	Advanced	Expert
Change	SDK	Apps	Nothing

Table 6: Comparing SecureOAuth to the other solutions.

CHAPTER 5: SECURITY CUE DESIGN USER STUDIES

5.1 In-Lab User Study

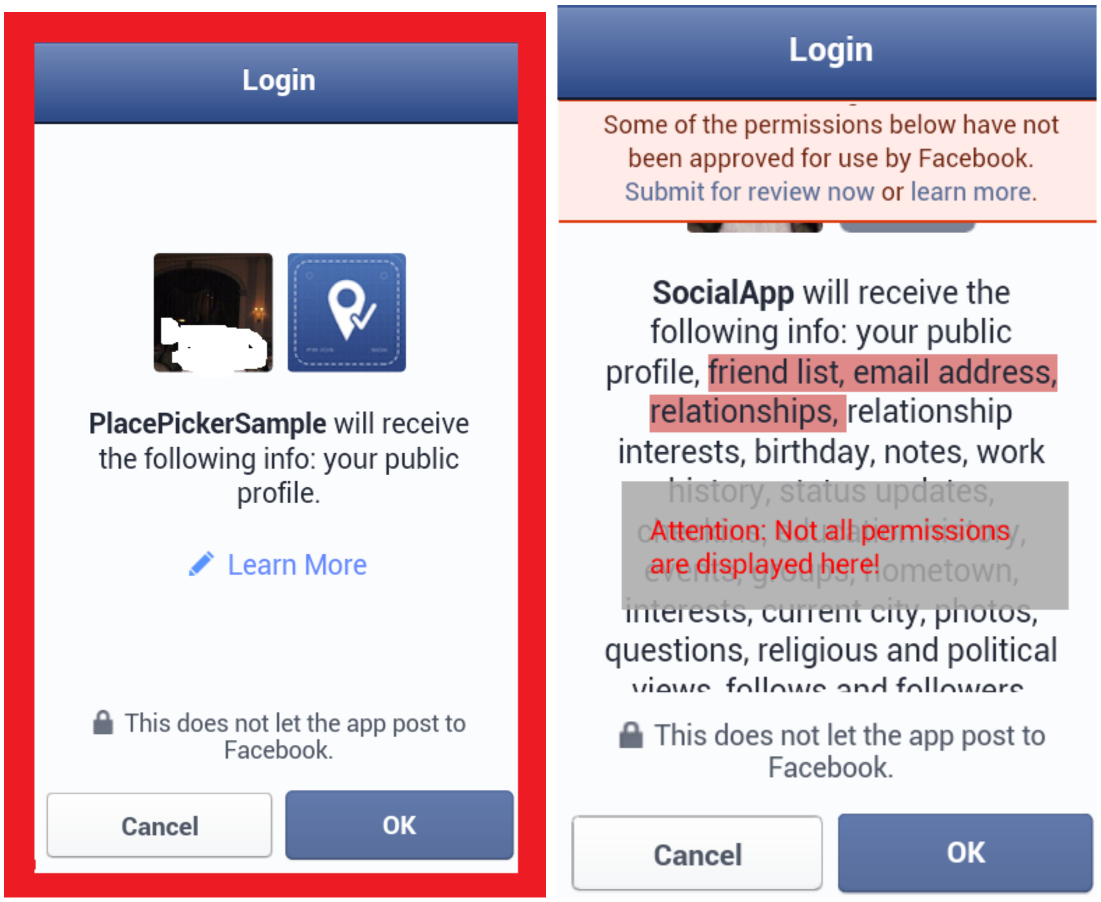
The WhiteList solution, discussed in Section 4.3, allows JavaScript injection in any page that are loaded in a WebView unless that page has been included in the protected URL list. The solution blocks any JavaScript injection attempt against these protected URLs without giving the user any feedback or a choice. In this chapter, we investigate the possibility of giving users feedback that can assist them take informed decisions. Now, whenever the system detects any of the attacks we discussed earlier in Section 3.1.2, it sends a security clue to the user who must then decide whether to proceed with the authentication and authorization steps or cancel them. The security clues can also lead users into uninstalling the application and/or giving it bad review. Thus, the security clues must be designed carefully to help users take the right decisions. For this purpose, we have conducted a user study that proposes different security clues‘ designs and for each design, it measures it’s noticeability, understandability, and effectiveness. Moreover, the study evaluate participants‘ separation understanding between the mobile app context and provider context, in cases where the mobile apps communicate with online service providers (e.g. Facebook, Twitter, and Dropbox).

Scenario Code	**Page Target**	**Description**
A0	Authentication	No Change
A1	Authentication	Red Border + Message
A2	Authentication	Red Background (Input Fields) + Message
A3	Authentication	Toasting the password
B0	Authorization	No Attack
B1	Authorization	Attack with no visual feedback
B2	Authorization	Red Border
B3	Authorization	Highlight the permissions + Toast Message

Table 7: The eight scenarios that are used in designing the study testing applications.

Test App Code	**Scenario Code 1**	**Scenario Code 2**
T1	A0	B0
T2	A1	B0
T3	A2	B0
T4	A3	B0
T5	A0	B1
T6	A0	B2
T7	A0	B3

Table 8: The seven testing applications used in our user study.


Login

PlacePickerSample will receive the following info: your public profile.

Learn More

This does not let the app post to Facebook.

Cancel OK

(a) First Warning

Login

Some of the permissions below have not been approved for use by Facebook. Submit for review now or learn more.

SocialApp will receive the following info: your public profile, friend list, email address, relationships, relationship interests, birthday, notes, work history, status updates,

Attention: Not all permissions are displayed here!

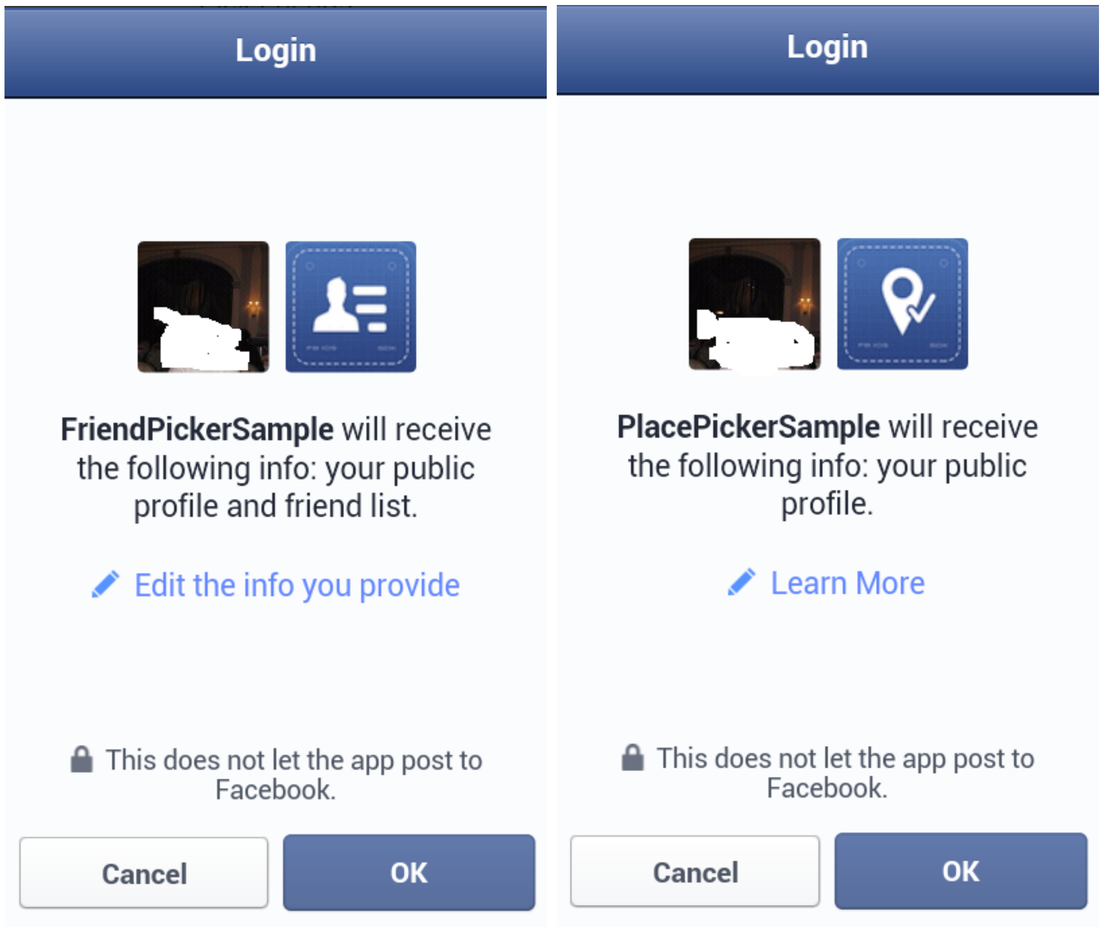
interests, current city, photos, questions, religious and political

This does not let the app post to Facebook.

Cancel OK

(b) Second Warning

Figure 17: Using different warning elements on the Authorization page: In a, the red border is used. In b, the highlight on portion of the permissions and a toast message are used.


Login

FriendPickerSample will receive the following info: your public profile and friend list.

Edit the info you provide

This does not let the app post to Facebook.

Cancel OK

(a) Normal Authorization Page

Login

PlacePickerSample will receive the following info: your public profile.

Learn More

This does not let the app post to Facebook.

Cancel OK

(b) Manipulated Authorization Page

Figure 18: The first image shows a normal authorization page, the second image shows a manipulated authorization page where one permissions is been taken out.

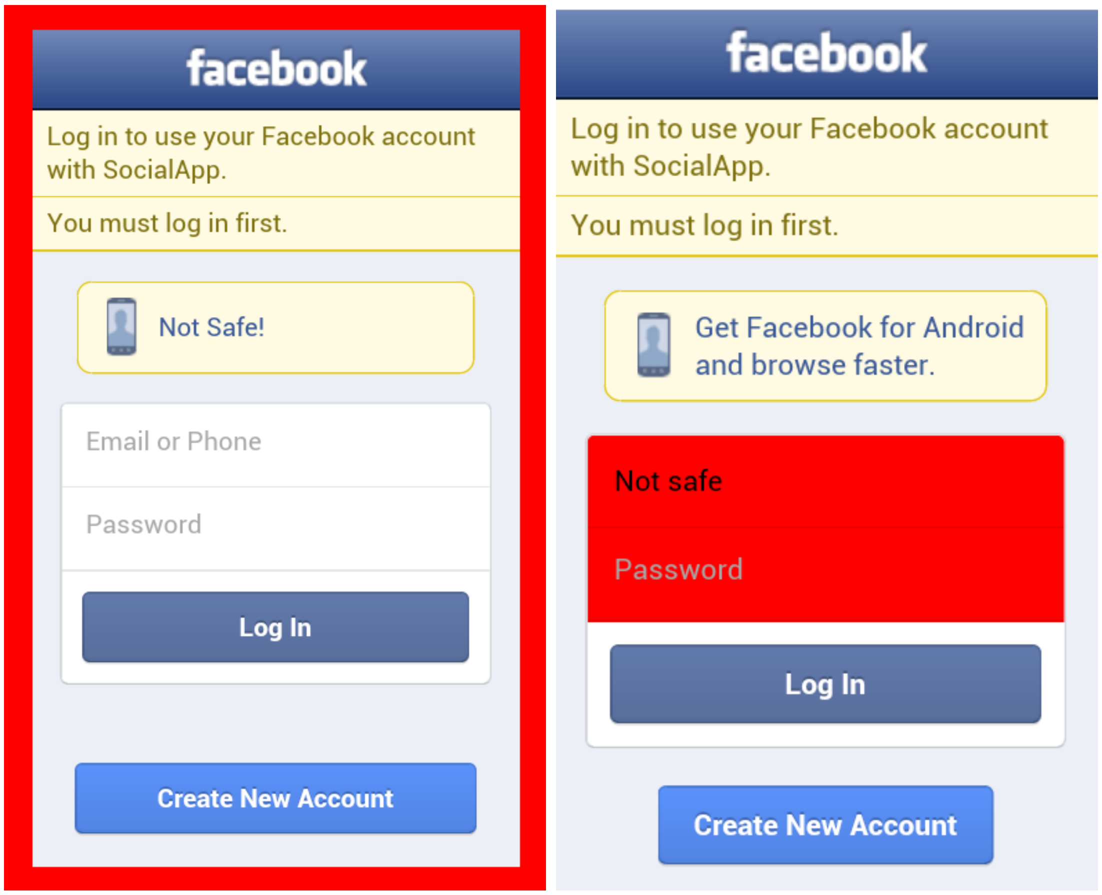
5.1.1 User Study Description

In our study, seven different Android applications are developed to be used as a part of the user experiment. Each application contains different security clue designs which are to be measured in terms of noticeability, understandability, and effectiveness. Each participant is asked to install these applications into a lab Android device, then, run, authenticate and authorize the app into their Facebook profiles. Each one of these apps contains the Facebook Android SDK that implements OAuth 2.0. As part of the OAuth implementation, the app displays Facebook authentication and authorization pages, see Figures 20(a) and 18(a) respectively. The SDK uses a WebView as it's default method for displaying these pages. We enabled the JavaScript on these WebViews to inject the scripts that would show the different security clues.

5.1.2 User Study Design

Table 7 shows the eight different scenarios for displaying the authentication and authorization pages. For the authentication page, there are four scenarios: the normal case and three different alert designs. For the authorization page, there are four scenarios: the normal case, two different alert designs, and B1 case where we change the permissions list but without displaying any alert messages.

Table 8 shows the seven testing applications generated from the different eight scenarios. For instance, in application T3, the authorization page is kept normal where as the authentication page is modified to have the two input fields (username and password) with red background plus a "Not Safe!" message being displayed inside the username input field. Overall, each application contains at most one modification



Log in to use your Facebook account with SocialApp.

You must log in first.

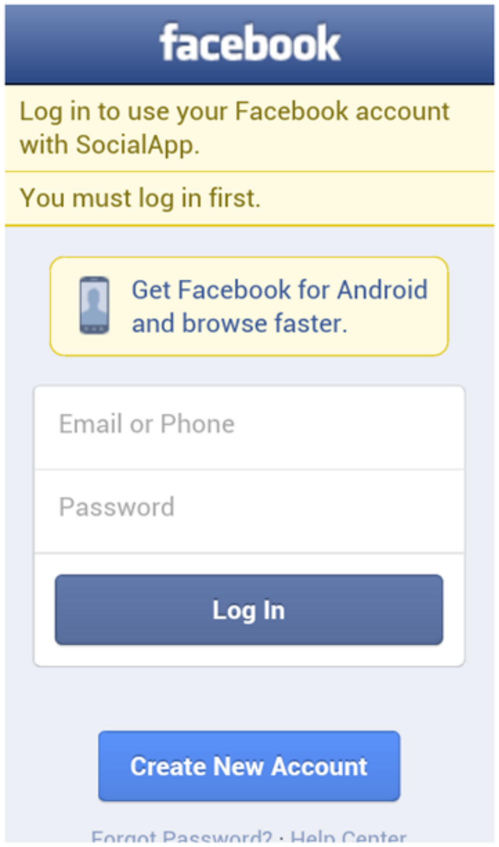
Log in to use your Facebook account with SocialApp.

You must log in first.

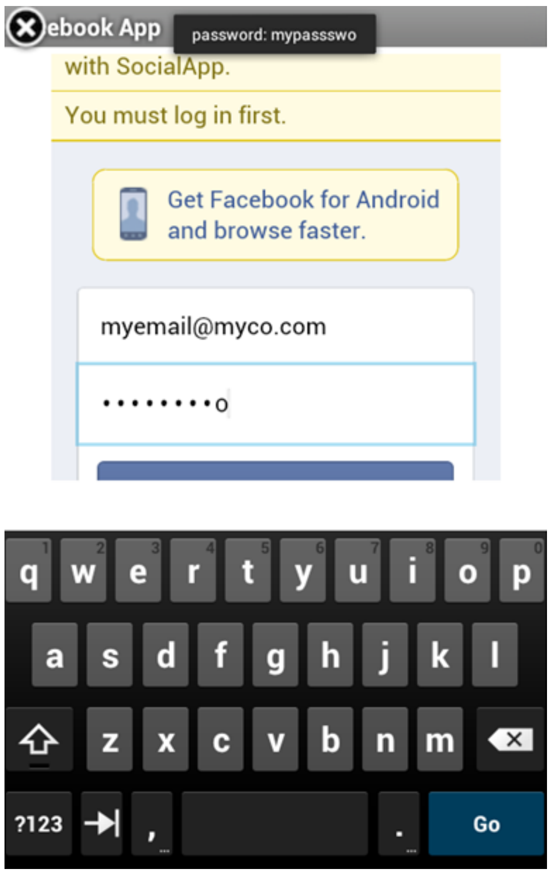
Get Facebook for Android and browse faster.

(a) First Warning (b) Second Warning

Figure 19: Using different warning elements on the Authentication page: In a, the red border and the text ”Not Safe” are used. In b, red back ground for the input fields and the text ”Not Safe” are used.


facebook
Log in to use your Facebook account with SocialApp.
You must log in first.
Get Facebook for Android and browse faster.
Email or Phone
Password
Log In
Create New Account
Forgot Password? · Help Center

(a) Normal Authentication Page



(b) Hacked Authentication Page

Figure 20: The figure in a and b shows a normal authentication page and a hacked authentication page respectively. In b, the app reads user password and display it in a toast message.

been done to either the authentication page or the authorization page. The study starts by asking the participants to install T1 into the lab Android device, following that, the participants need to run, authenticate, and authorize the app on his or her Facebook profile. After that the participant is asked to complete a pre-survey which contains nine simple questions regarding their age, gender, education, and Android usage behavior. Next, the participants are asked to repeat the above steps for the next three apps (T2, T3, and T4) then complete post survey 1. Following that, the participants are asked to install the remaining apps (T5, T6, and T7) then complete post survey 2. Finally, the participants are asked to complete a short survey concerning their belief about the separation between mobile apps and service providers.

5.1.3 Participants

Recruiting participants for our user study was done via sending emails and posting on social media. A call for participation email was sent to our school‘s graduate and undergraduate students. The email contains the study description and requirements for participation, which are: Participants must be at least 18 years old, have a Facebook account, and have used an Android device for at least 2 months. We had a total of 25 participants, 31% Female and 69% Male. In terms of familiarity with Android, 24% have been Android users for less than 1 year, 34% 2 to 3 years, 28% 3 to 4 years, and 14% greater than 4 years.

5.1.4 User Study Results

The aim of this study is to finalize a security clue design that is easy to notice, comprehensible, and effective. For the authentication page, three designs were selected: red border and an embedded message, red background for the input fields and a message, and toasting the password as participant enters it. For the authorization page, two designs were selected: red border, highlights in addition to a toast message. For each one of these designs we asked participants to answer a set of questions that tend to capture their reaction and feedback. The security clue designs were generated by injecting a JavaScript into the WebViews that displayed the authentication and authorization pages. The injections were done after loading the two pages by overriding the *onPageFinished()* method of the WebView class.

Authentication Page

We used four different authentication page designs to test user awareness, concerns, noticeability, and understandability of the authentication process. Moreover, we test the effectiveness of the different alerts that we integrated into the authentication pages.

1. Effectiveness: A one-way repeated measured analysis of variance (ANOVA) was conducted to evaluate the null hypothesis that there is no change in participants' reluctance to enter their credentials when measured before displaying the security clues and after displaying three different security clues (N=29). The results of the ANOVA indicated a significant security clue effect, $p < 0.05$ [Wilks' Lambda = 0.62 , F(3,26) = 5.33, p = 0.005, n^2 = 0.38]. Thus, there is significant evidence to reject the null hypothesis.

Follow up comparisons indicated that not each pairwise difference was significant. There was a significant increase in the reluctance to provide the credentials between the base case page and the rest of pages; however, there was no significant difference

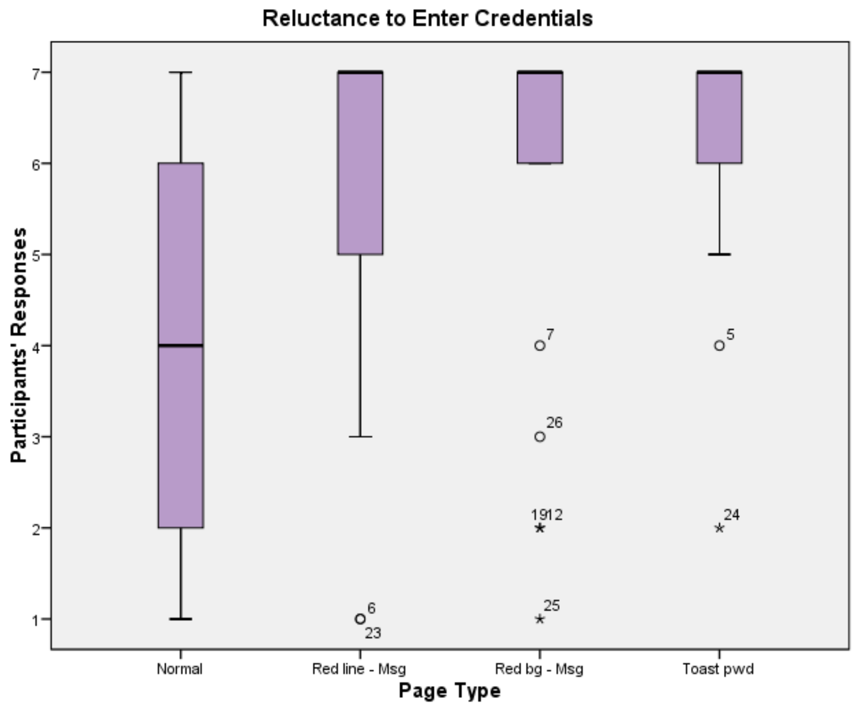


Figure 21: Reluctance to Enter Credentials.

between the three security clue designs, see Table 9. Figure 21 shows the distribution of participants‘ responses for the four cases. Showing the security clues increased participants‘ reluctance to enter their credentials.

	Normal	Redline	Redbg	Toast
Normal		0.02	0.009	0.002
Redline			1.00	1.00
Redbg				1.00

Table 9: Pairwise Comparisons Results - Reluctance to Enter Credentials.

2. Noticeability: A one-way repeated measured analysis of variance (ANOVA) was conducted to evaluate the null hypothesis that there is no difference in participants‘ noticeability for the different security clues (N=29). The results of the ANOVA indicated a significant difference in terms of noticeability for the different security

clues, $p < 0.05$ [Wilks‘ Lambda = 0.68 , F(3,26) = 4.02, p = 0.02, $n^2 = 0.32$]. Thus, there is significant evidence to reject the null hypothesis.

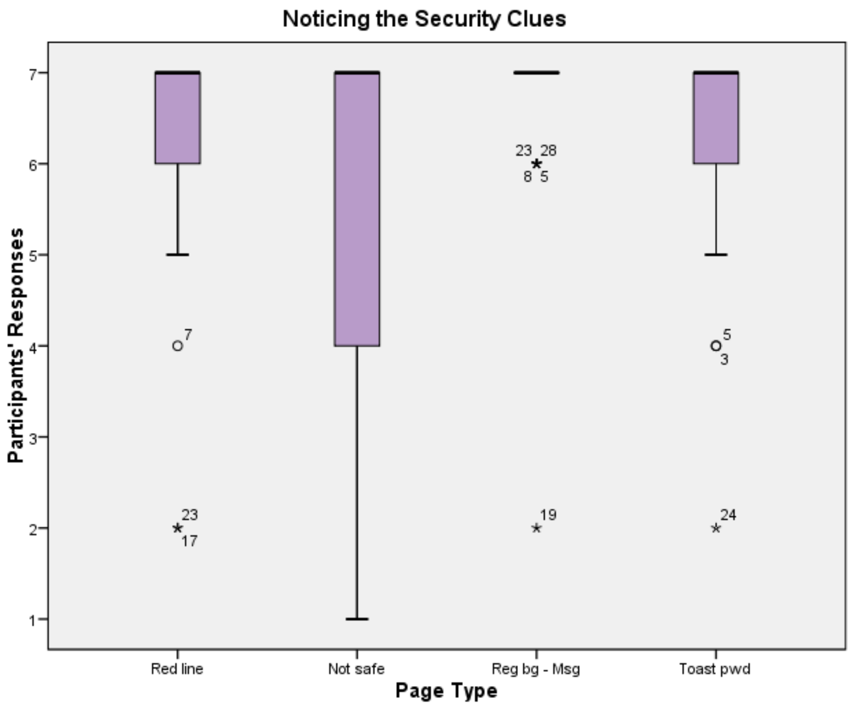


Figure 22: Noticing the Security Clues.

Follow up comparisons indicated that not each pairwise difference was significant as shown in Table 10. For instance, there was a significant difference between the noticeability of redline and message design and red background and message design. By looking at the mean values, participants stated that the red background and message design is more noticeable than the redline and message design. Having a closer look at the redline and message design, participants found that noticing the redline easier than noticing the message. Figure 22 depicts the participants‘ responses with regard to the noticeability.

	Redline	Not Safe	Redbg	Toast
Redline		0.11	1.00	1.00
Not Safe			0.02	0.50
Redbg				0.54

Table 10: Pairwise Comparisons Results - Noticing the Security Clues.

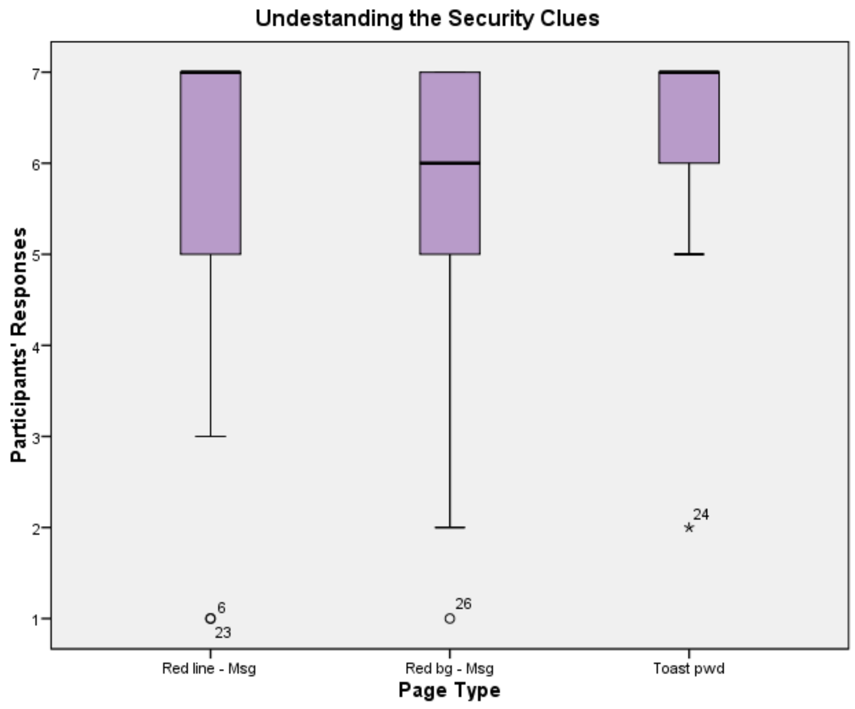


Figure 23: Understanding the Security Clues.

3. Understandability: A one-way repeated measured analysis of variance (ANOVA) was conducted to evaluate the null hypothesis that there is no difference in participants‘ understandability for the different security clues (N=29). The results of the ANOVA indicated a significant difference in terms of noticeability for the different security clues, $p < 0.05$ [Wilks‘ Lambda = 0.79 , F(2,27) = 3.64, p = 0.04, $n^2 = 0.21$]. Thus, there is significant evidence to reject the null hypothesis. Follow up comparisons indicated none of the pairwise difference was significant as shown in Table 11. Figure 23 displays the participants‘ responses with regard to the understandability

	Redline	Redbg	Toast
Redline		1.00	0.20
Redbg			0.06

Table 11: Pairwise Comparisons Results - Understanding the Security Clues.

	Applications	Facebook	System
Applications		0.12	0.0001
Facebook			0.045

Table 12: Pairwise Comparisons Results - Who Generates Authentication Page.

measure.

4. Generate Authentication Page: A one-way repeated measured analysis of variance (ANOVA) was conducted to evaluate the null hypothesis that there is no difference in participants‘ assumptions regarding who generates the authentication page (N=29). The results of the ANOVA indicated a significant difference in participants‘ assumptions with regard to who generates the authentication page: Android system, Applications, and Facebook, $p < 0.05$ [Wilks‘ Lambda = 0.54 , F(2,27) = 11.31, p = 0.0001, n^2 = 0.46]. Thus, there is significant evidence to reject the null hypothesis.

Follow up comparisons indicated that not each pairwise difference was significant according to Table 12. For example, there was a significant difference between participants‘ assumption of that Apps generate the login page and that the system generates it. Figure 24 shows the participants‘ responses about who generates the authentication page.

5. Generate Redline-Message and Red Background-Message: A one-way repeated measured analysis of variance (ANOVA) was conducted to evaluate the null hypoth-

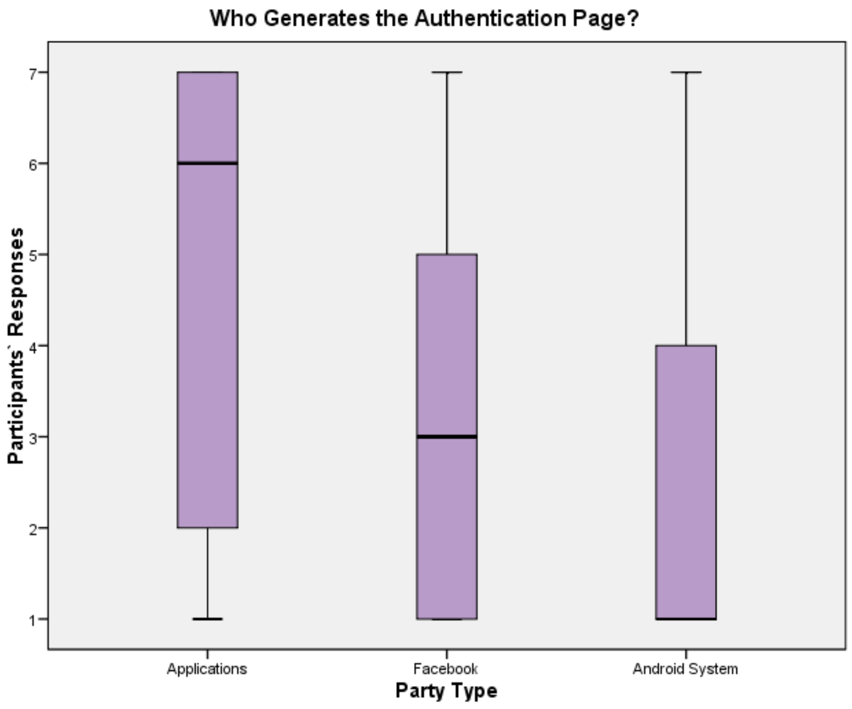


Figure 24: Participants' Assumptions with Regard to Who Generates the Authentication Page.

esis that there is no difference in participants' assumptions regarding who generates the security clue, redline and message(N=29). The results of the ANOVA indicated no significant difference, thus, we accepted the null hypothesis.

5. Generate Red Background-Message: A one-way repeated measured analysis of variance (ANOVA) was conducted to evaluate the null hypothesis that there is no difference in participants' assumptions regarding who generates the security clue, red backgrounds and message(N=29). The results of the ANOVA indicated no significant difference, thus, we accepted the null hypothesis.

6. Toast Password: A one-way repeated measured analysis of variance (ANOVA) was conducted to evaluate the null hypothesis that there is no difference in participants' assumptions regarding who toasts the password (N=29). The results of the

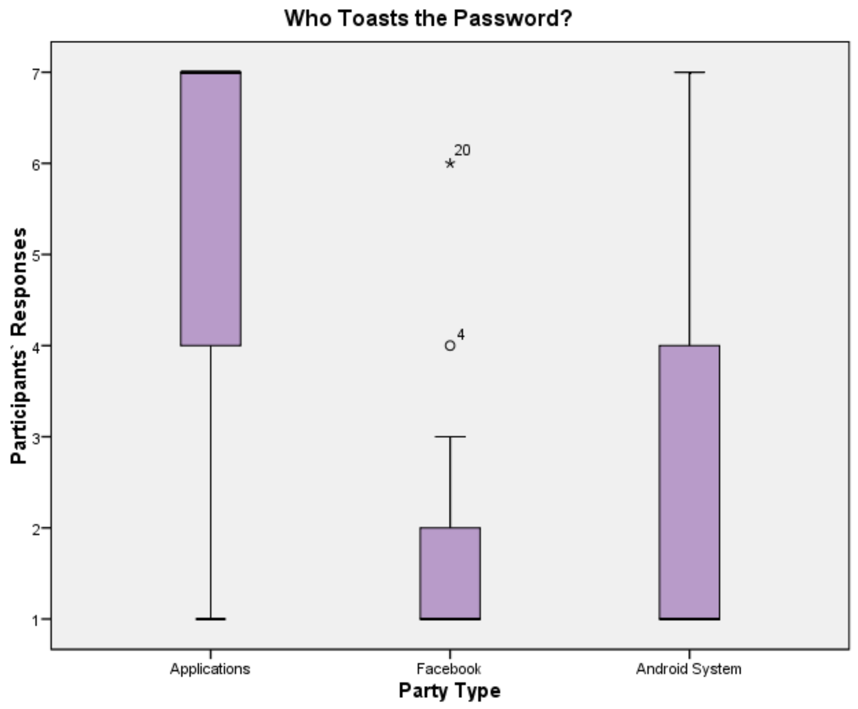


Figure 25: Participants‘ Assumptions with Regard to Who Toasts the Password.

ANOVA indicated a significant difference in participants‘ assumptions for the three choices: Android system, Applications, and Facebook, $p < 0.05$ [Wilks‘ Lambda = 0.31 , F(2,27) = 30.3, p = 0.0001, n^2=0.70]. Thus, there is significant evidence to reject the null hypothesis.

Follow up comparisons indicated that not each pairwise difference was significant as depicted in Table 13. For instance, there was a significant difference between participants‘ assumption of that Apps toast passwords and that the Facebook toasts it. Moreover, there was a significant difference between participants‘ assumption of that Apps toast passwords and that the system toasts them. However, There was no significant difference between participants‘ assumption of that Facebook toasts passwords and that the system toasts them. Figure 23 shows the participants‘ responses

	Applications	Facebook	System
Applications		0.0001	0.001
Facebook			0.07

Table 13: Pairwise Comparisons Results - Who Toasts Password.

about who toasts their passwords.

Authorization Page

In this study, we used four different authorization page designs. The baseline authorization page design in which we changed nothing, the manipulated authorization page design in which we removed one sensitive permission but without giving any alert. The other two designs were: an authorization page with a redline border, and an authorization page in which some of the permissions were highlighted and a toast message was displayed. In this part of the study, we looked at four factors: noticeability, understandability, and effectiveness. Besides, we looked at users' perception on the source of the authorization page content.

1. Effectiveness: A one-way repeated measured analysis of variance (ANOVA) was conducted to evaluate the null hypothesis that there is no difference in participants' willingness to authorize an applications with and without adding security clues (N=29). The results of the ANOVA indicated no significant difference, thus, we accepted the null hypothesis.

2. Understandability: A one-way repeated measured analysis of variance (ANOVA) was conducted to evaluate the null hypothesis that there is no difference in participants' understanding of the security clues that were shown on the authorization page

	Applications	Facebook	System
Applications		0.53	0.04
Facebook			0.82

Table 14: Pairwise Comparisons Results - Who Generates Authorization Page.

(N=29). The results of the ANOVA indicated no significant difference, thus, we accepted the null hypothesis.

3. Noticeability: A one-way repeated measured analysis of variance (ANOVA) was conducted to evaluate the null hypothesis that there is no difference in participants‘ noticeability for the two security clue designs (N=29). The results of the ANOVA indicated no significant difference, thus, we accepted the null hypothesis.

4. Generate Authorization Page: A one-way repeated measured analysis of variance (ANOVA) was conducted to evaluate the null hypothesis that there is no difference in participants‘ assumptions regarding who generates the authorization page (N=29). The results of the ANOVA indicated a significant difference in participants‘ assumptions with regard to who generates the authorization page: Android system, Applications, and Facebook, $p < 0.05$ [Wilks‘ Lambda = 0.79 , F(2,27) = 3.64, p = 0.04, n^2 = 0.21]. Thus, there is significant evidence to reject the null hypothesis.

Follow up comparisons indicated that not each pairwise difference was significant as shown in Table 14. There was a significant difference between participants‘ assumption of that Apps generate the authorization page and that the system generates it. Figure 26 shows the participants‘ responses about who generates the authorization page.

Separation: The use of OAuth has helped developers to overcome the trust prob-

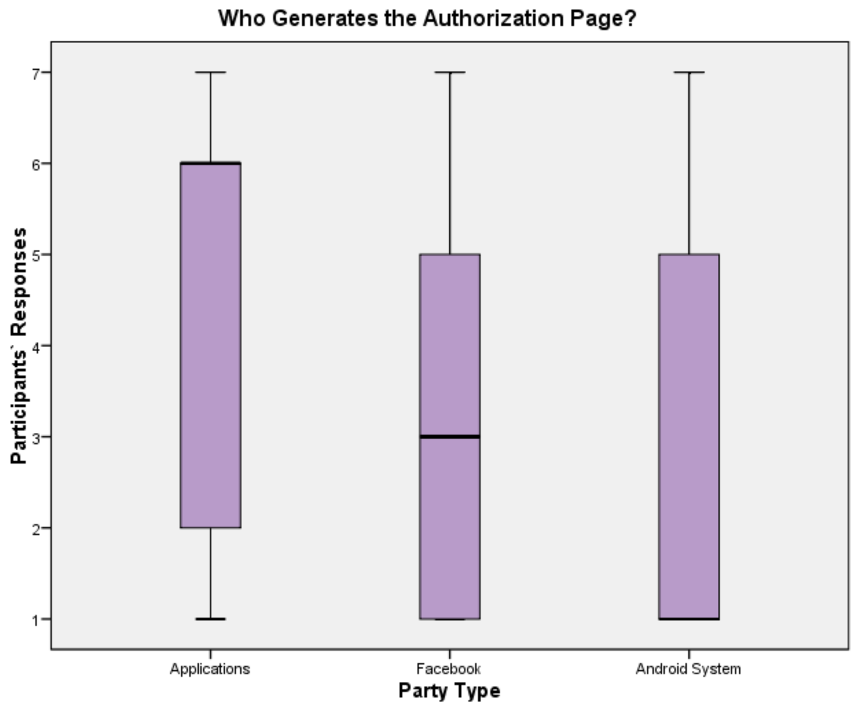


Figure 26: Participants‘ Assumptions with Regard to Who Generates the Authorization Page.

lem in which users felt reluctant to give away their credentials to third-party clients. OAuth recognizes two clients types: confidential and public clients [56]. Confidential clients are capable of maintaining the confidentiality of their credentials such as web applications . Public clients are incapable of maintaining the confidentiality of their confidentiality such as user-agent-based applications (e.g. browser plugins) and native applications (e.g. mobile applications). Web applications have been around longer, thus, users have established more trust on them than mobile applications. In addition, OAuth have different implementations choices in case of mobile applications as we stated in Section 1.1.1 which may affect users experience, comprehension and trust. For this reason, we devoted part of the study to investigate users‘ beliefs and assumptions in regard to OAuth in web applications versus mobile applications. By

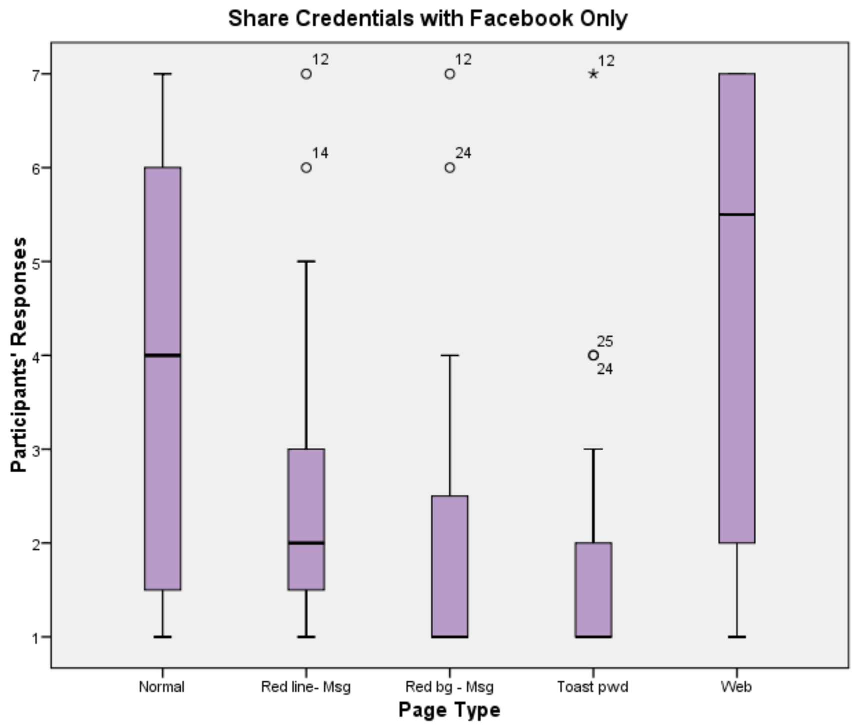


Figure 27: Sharing credentials with Facebook only.

the end of the study, we asked the participants the following set of questions:

Facebook Only: A one-way repeated measured analysis of variance (ANOVA) was conducted to evaluate the null hypothesis that there is no difference in participants‘ believe in that once they authenticate, they only share their credentials with Facebook only (N=29). The results of the ANOVA indicated a significant difference in participants‘ believe for the five choices: plain authentication page, three modified authentication pages, and Facebook login prompt on the web, $p < 0.05$ [Wilks‘ Lambda = 0.40 , F(4,24) = 9.10, p = 0.0001, n^2=0.60]. Thus, there is significant evidence to reject the null hypothesis.

Follow up comparisons indicated that not each pairwise difference was significant

	Normal	Redline	Redbg	Toast	Web
Normal		0.08	0.005	0.001	1.00
Redline			0.33	0.02	0.01
Redbg				0.60	0.000
Toast					0.000

Table 15: Pairwise Comparisons Results - Sharing with Facebook Only.

	Normal	Redline	Redbg	Toast	Web
Normal		0.24	0.05	0.11	0.25
Redline			1.00	1.00	0.005
Redbg				1.00	0.001
Toast					0.004

Table 16: Pairwise Comparisons Results - Sharing with Facebook and App/Website.

as shown in Table 17. Figure 27 shows the participants‘ responses with the regard to whether Facebook is the only party that participants are sharing their credentials with after the authentication process.

Facebook and Mobile App or Website: A one-way repeated measured analysis of variance (ANOVA) was conducted to evaluate the null hypothesis that there is no difference in participants‘ believe in that once they authenticate, they share their credentials with Facebook and the App (N=29). The results of the ANOVA indicated a significant difference in participants‘ believe for the five choices: normal mobile Facebook authentication page, three modified mobile Facebook authentication pages, and Facebook login prompt on the web, $p < 0.05$ [Wilks‘ Lambda = 0.54 , F(4,24) = 5.04, p = 0.04, n^2 = 0.46]. Thus, there is significant evidence to reject the null hypothesis.

Follow up comparisons indicated that not each pairwise difference was significant as shown in Table 17. For instance, there was no significant difference between the normal mobile Facebook authentication page and the Facebook login prompt on the

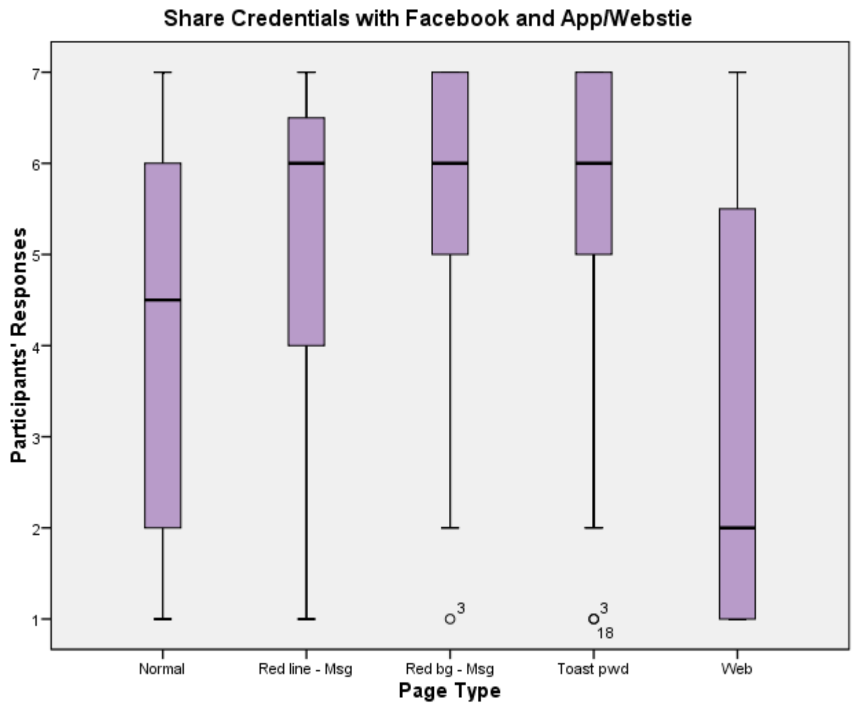


Figure 28: Sharing credentials with Facebook and App/Website.

web. However, there was a significant difference between the three modified mobile Facebook authentication pages and the Facebook login prompt on the web.

The Website and Moobile App Can Access: A one-way repeated measured analysis of variance (ANOVA) was conducted to evaluate the null hypothesis that there is no difference in participants‘ believe in that a mobile application or a website can access the content of the Facebook authentication page and the data they enter into that page (N=29). The results of the ANOVA indicated a significant difference in participants‘ believe for the five choices: normal mobile Facebook authentication page, three modified mobile Facebook authentication pages, and Facebook login prompt on the web, $p < 0.05$ [Wilks‘ Lambda = 0.37 , F(4,24) = 10.10, p = 0.0001, $n^2 = 0.63$]. Thus, there is significant evidence to reject the null hypothesis.

	Normal	Redline	Redbg	Toast	Web
Normal		0.03	0.003	0.002	0.04
Redline			0.48	0.16	0.000
Redbg				1.00	0.000
Toast					0.000

Table 17: Pairwise Comparisons Results - App / Website can access content and data.

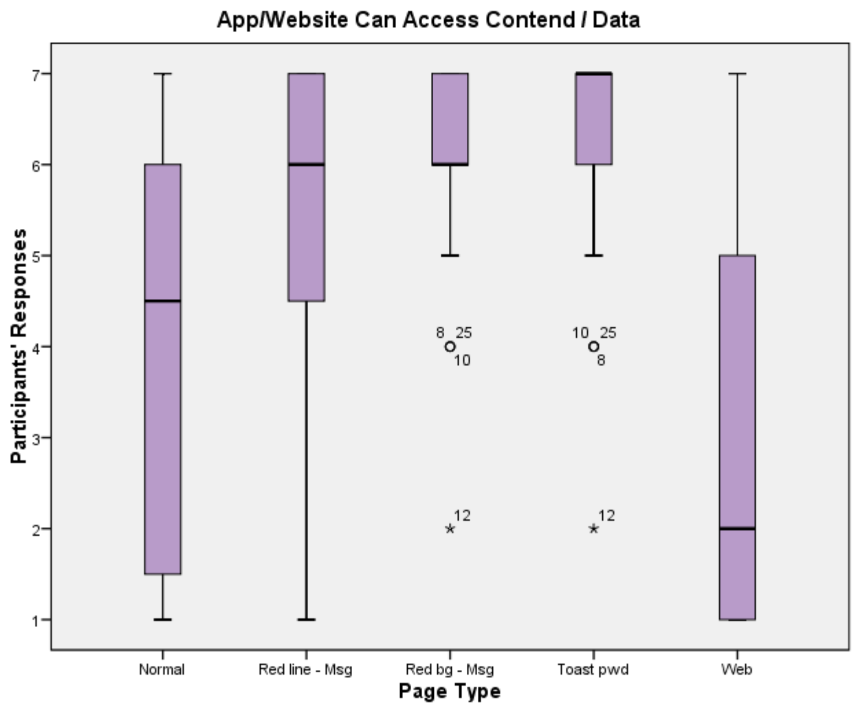


Figure 29: App / Website can access content and data.

Follow up comparisons indicated that not each pairwise difference was significant as shown in Table 17. For instance, there was no significant difference between the three modified mobile Facebook authentication pages. However, there was a significant difference between the normal mobile Facebook authentication page and the three modified mobile Facebook authentication pages. Moreover, there was a significant difference between the normal mobile Facebook authentication page and the Facebook login prompt on the web.

Discussion

Our results showed that the security clue designs that we chose were effective in increasing participants‘ reluctance to authenticate and authorize the applications on their Facebook profiles. The designs were so close to each other in terms of noticeability and understandability according to the significant tests that we ran. The red background fields and a message design was the most noticeable one and toasting the password design was the most understandable design. Some participants agreed with the previous result; however, they stated that toasting the password might leak their passwords to whoever setting next to them and looking at their screens.

Most participants believed that mobile applications actually generate the authentication and authorization pages not the Facebook. We also found that participants have more trust in web applications than in mobile applications when authentication is involved. They believed that authenticating to an application can leak their credentials to other parties. We noticed that their suspicious beliefs increased when we used the different security clue designs. Participants also believed that mobile applications have access not only to the credentials but also to the content of the authentication and authorization pages.

We found that participants lack the separation understanding between the phone and Facebook profile for the mobile applications. For example, the majority of participants failed to locate the applications on their Facebook profile after authenticating and authorizing them. Moreover, the majority of participants could not differentiate between removing the applications from the phone verses removing the application from their Facebook profile. This calls for the need of educating users about security awareness.

The summary of our user study was that users have low trust in mobile applications. The low trust increases with the applications that involve authentication and authorization. Because of that, users are expected to react to the security clues that we send and most likely to take immediate action, such as canceling the authorization and/or uninstalling the application from the device.

5.2 Online User Study - Security Cue Designs for OAuth-WebView Attacks

In this chapter we discuss our large scale user study that we conducted to compare between seven different security cue designs. We came up with six of these designs and borrowed one from the literature. We describe these designs, the user study settings, and the results.

5.2.1 Study Description

The purpose of the user study is to choose the most noticeable and understandable security cue design that can be used to alert mobile phone users when an attack against their privacy is being caught. In previous chapters we talked about different kind of attacks that target the vulnerabilities of WebView particulary when it is used to implement the OAuth. Although our proposed solutions have the choice of blocking the attacks once detected. In some scenarios it is more prominent to leave the decision to the end user. Moreover, even if the solution blocked the attack, the user must be notified. Thus, this user study is meant to find good security cue design to serve both purposes. In doing so, we came up with six security cue designs and borrowed one from the literature. We developed different Android applications and embedded the security cue designs in them. The applications has an OAuth-WebView implementation to connect them to Facebook. The security cue designs were inserted in two main pages: the authentication and authorization pages. We added hooks inside these applications to generate participation codes and to track participants‘ interactions with the different pages. The applications have been then uploaded to an online emulator service called *appetize.io*. Participants were asked to use the mobile applications that are running on the online emulator then to complete a survey that is hosted on Google Docs. The actual purpose of the study was kept hidden from the participants. Instead participants were told that the purpose of the study is to help Android developers and researchers to choose the right platforms and tools for testing their products. Moreover, it helps service providers enhance their tools and platforms. At a certain point in the survey, participants were told the actual purpose of the study. The survey included questions about the pages that they have seen while using the applications, if they have noticed any abnormal components, the decision they made, and why did they made those decisions.

5.2.2 Participants

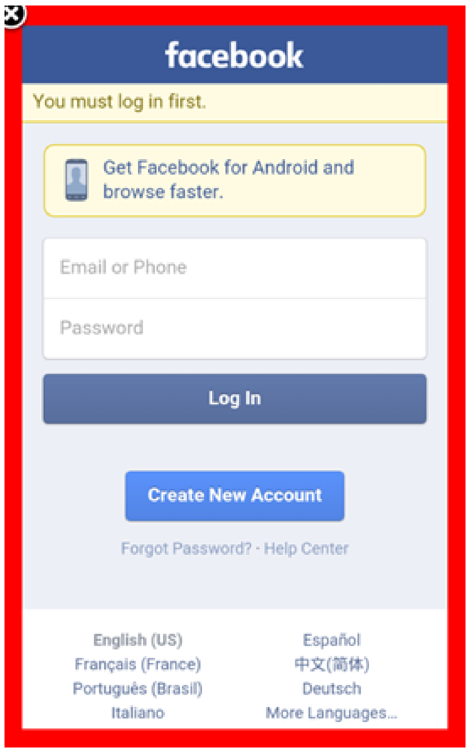
Recruitment of participants was done by Amazon Turk [1] with level of confidence of 90-95%, we paid each participant \$0.25. We had 400 participants use and complete the survey. Participation in the study should take 10-15 minutes. Participants will be asked to use a mobile applications running on the online Android emulator platform called appetizer and complete an online survey. Upon completion of the study, if you authorize the necessary app(s) on your Facebook account, you may remove them by going to Settings then Apps on your Facebook, selecting the appropriate app, and pressing the Remove button. Eligibility: Eligible participants will be at least the age of 18, Android user, and has Facebook account.

5.2.3 Security Cue Designs

We have seven security cue designs used in the authentication and authorization pages. Four of them were used in the authentication page and three used in the authorization page. Six of these designs are created by us and we borrowed one from the literature.

5.2.4 Authentication Page - Security Cue Designs

Four security cure designs were studied at the authentication page with three of those were designed by us and the fourth was borrowed from the literature. The designs are shown below: Design A uses a red background and a message, Design B uses the red X sign, Design C uses a toast to display the password as the user enters it, and Design D uses a red line around the authentication page.


facebook

You must log in first.

Get Facebook for Android and browse faster.

Email or Phone

Password

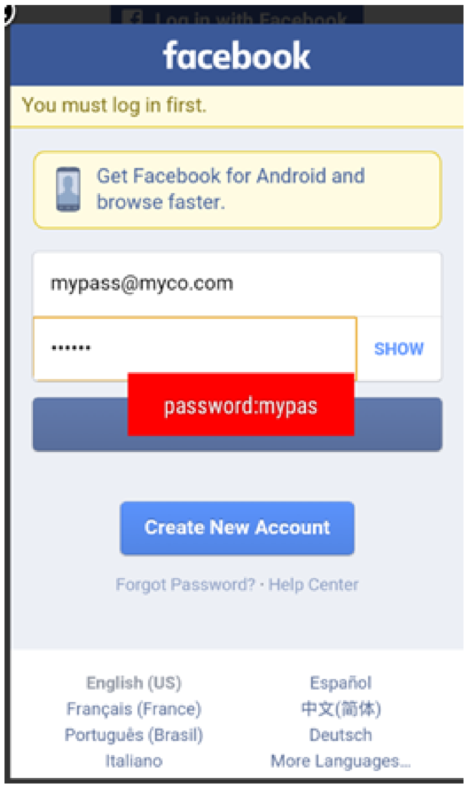
Log In

Create New Account

Forgot Password? · Help Center

English (US) Español
Français (France) 中文(简体)
Português (Brasil) Deutsch
Italiano More Languages...

Figure 30: Design D uses a red line around the authentication page.


facebook

You must log in first.

Get Facebook for Android and browse faster.

mypass@myco.com

••••••

SHOW

password:mypas

Create New Account

Forgot Password? · Help Center

English (US) Español
Français (France) 中文(简体)
Português (Brasil) Deutsch
Italiano More Languages...

Figure 31: Design C toasts the password as the user enters it.

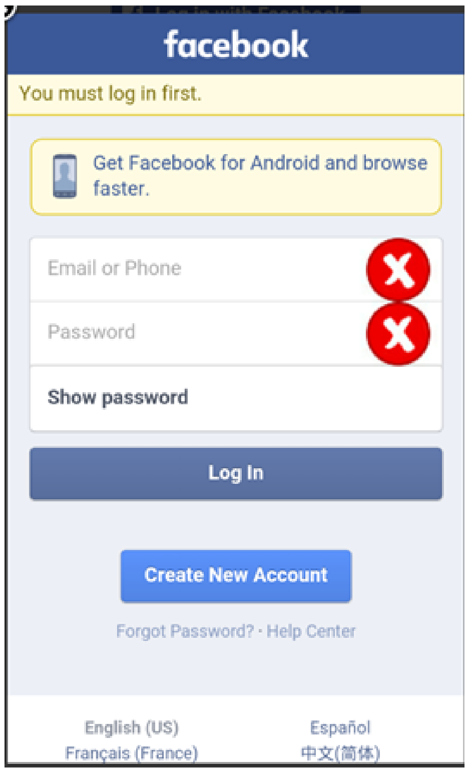


Figure 32: Design B uses the red sign (Traffic Light).

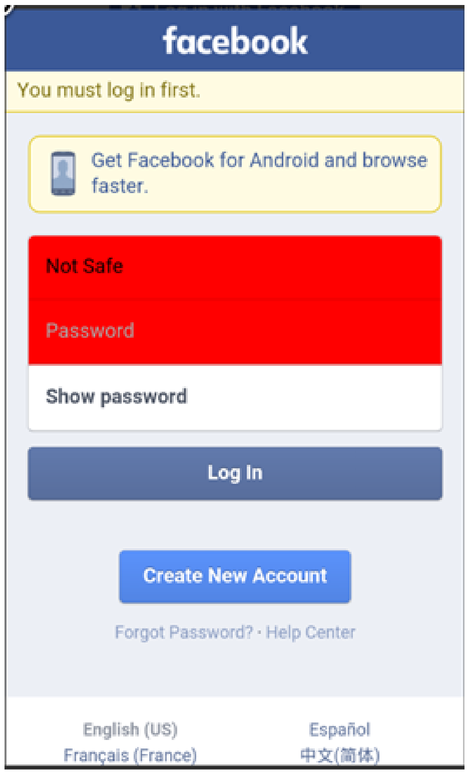


Figure 33: Design A uses red backgrounds for the input fields and a message).

5.2.5 Authentication Page - Survey

We asked participants to complete a survey after using our application. The survey is composed of 22 questions: 14 questions are about the security designs, 4 questions about their mobile device usage pattern, and 4 demographic questions.

1. Security Cue Designs

   (a) Did you log into a Facebook account from the Platforms Tester application?

   - Yes.
   - No.

   (b) Which account did you use to log into Facebook?

   - The study Facebook account (joneyj88@gmail.com).
   - My Own Facebook account.

   (c) Please select the option that most accurately completes the following sentence: "I logged into Facebook using my own Facebook account because..."

   - I didn't see anything suspicious.
   - I normally log into Facebook from mobile apps.
   - It was faster/easier to use my own Facebook account.
   - I chose to, there was no particular reason.
   - other.

   (d) Please select the option that most accurately completes the following sentence: "I logged into Facebook using the study Facebook account because..."

- I do not trust third-party applications.
- I felt more comfortable using the study account.
- I chose to, there was no particular reason.
- I saw something suspicious.
- other.

(e) Please select the option that most accurately completes the following sentence: "I didn't log into Facebook from the Platforms Tester app because..."

- I do not trust third-party applications.
- I chose not to.
- I wanted to finish early.
- I saw something suspicious on the page.
- other.

(f) Please describe the suspicious elements that you saw on the authentication (login) page?

(g) Which one of these designs did you see when you used the Platfroms Tester App?

- Design A.
- Design B.

- Design C.
- Design D.
- None.

(h) Which one of these designs is the most visible (noticeable)?

- Design A.
- Design B.
- Design C.
- Design D.

(i) Which one of these designs is the most understandable?

- Design A.
- Design B.
- Design C.
- Design D.

(j) Did you authorize the Platforms Tester app on your Facebook account?

- Yes.
- No.

(k) Why didn't your authorize the Platforms Tester app on your Facebook account?

- Because i do not trust third-party mobile applications.
- Because i chose not to.
- I wanted to finish early.

- Because i wanted to finish the user study early.
- other.

(l) Why did you authorize the Platforms Tester app on your Facebook account?

- Because I normally authorize third-party apps on my social accounts such as Facebook.
- Because I did not see anything suspicious.
- I had to complete the user study.
- other.

(m) When entering your username and password in Page 1 (authentication) you are sharing your username and password with Facebook only.

- Entirely disagree (1 - 7) Entirely agree

(n) When entering your username and password in Page 1 (authentication) you are sharing your username and password with Facebook and the third-party mobile app.

- Entirely disagree (1 - 7) Entirely agree

(o) I think that mobile apps are able to access the contents Page 1(authentication) and the user entered data (username and/or password)

- Entirely disagree (1 - 7) Entirely agree

(p) I usually look carefully on the list of permissions that third-party mobile applications are requesting (Page 2) on my social accounts (e.g. Facebook)

- Entirely disagree (1 - 7) Entirely agree

(q) I feel not safe granting third-party mobile applications any privileges over my social accounts (e.g. Facebook)

- Entirely disagree (1 - 7) Entirely agree

(r) I think that mobile apps are able to access the contents of Page 2 (authorization).

- Entirely disagree (1 - 7) Entirely agree

(s) When entering your username and password in Page 2 (authentication) you are sharing your username and password with Facebook only.

- Entirely disagree (1 - 7) Entirely agree

(t) When entering your username and password in Page 2 (authentication) you are sharing your username and password with Facebook and the Doodle website.

- Entirely disagree (1 - 7) Entirely agree

(u) I think that the Doodle website is able to access the contents of Page 2 (authentication) and the user entered data such as the username and password.

- Entirely disagree (1 - 7) Entirely agree

(v) I usually look carefully on the list of permissions that a website such as Doodle is requesting on my social accounts (Page 3).

- Entirely disagree (1 - 7) Entirely agree

(w) I feel not safe granting websites such as Doodle any privilege over my social accounts (e.g. Facebook)

- Entirely disagree (1 - 7) Entirely agree

(x) I think that websites such as Doodle are able to access the contents of Page 3 (authorization). (for instance, it can manipulate the list of permissions)

- Entirely disagree (1 - 7) Entirely agree

2. Usage Pattern

(a) How often/frequent do you install third-party applications into your Android phones?

- Never (1 - 7) Every time

(b) How often/frequent do you use third-party apps to access your social accounts (e.g. Facebook, Twitter, etc.) in your Android device?

- Never (1 - 7) Every time

(c) Do you have an antivirus software on your Android device?

- Yes
- No

(d) How would you describe your proficiency level in Android devices?

3. Demographic

(a) What is your age?

- 18 to 24

- 25 to 34
- 35 to 44
- 45 to 54
- 55 to 64
- 65 to 74
- 75 or older

(b) What is your gender?

- Male
- Female

(c) What is the highest level of education you have completed?

- Some high school
- High school/GED
- Some college
- Associates degree
- Bachelors degree
- Masters degree
- Doctorate Degree
- Other
- Decline to answer

(d) What is your race/ethnicity?

- Asian/Pacific Islander

- Black/African-American
- White/Caucasian
- Hispanic
- Native American/Alaska Native
- Other/Multi-Racial
- Decline to answer

5.2.6 Participants Demography and Usage Pattern

The distribution of participants' demography ,Table 18, is as follows: of the 423 who used the app and completed the survey, 60% (240 participants) were Male and 40% (183 participants) were Female. When it comes to age, the majority (61%) were ages 25-34, (22%) were between 18-24. The remaining (17%) fell between the ages of 35-74. With regard to eduction, (38%) of them had completed an undergrade degree, whereas, (27%) had completed graduate degree, (28%) had completed some college or an associate degree, the remaining (7%) had either some high school or a high school/GED. Lastly, (80.5%) of participants were proficient in using Android phones, (12%) had average proficiency, and the remaining (7.5%) were low proficiency.

We asked participants three questions to capture their mobile phone usage pattern Table 19. The majority of participants 74% (344) had an antivirus installed on their devices, 26% (119) did not have an antivirus. With regard to apps installation habit: (61%) of participants install apps on their mobile devices very frequently, (19%) often installed , and (20%) installed them less often. More than half of participants (53%) frequently used third-party apps to access their online social accounts, (15%) often

Table 18: Participants Demography

Variable	n(%)
Gender	
Male	240(60)
Female	185(40)
Age	
18-24	104(22)
25-34	282(61)
35-44	56(12)
45-54	16(3)
55-74	7(2)
Education	
Some high school	6(1)
High school/GED	31(6)
Some college	89(19)
Associate‘s degree	40(9)
Bachelor‘s degree	174(38)
Master‘s degree	114(25)
Doctorate degree	11(2)
Proficiency in Android	
Level 1 (Very Basic)	5 (1)
Level 2	11(2)
Level 3	22(5)
Level 4	57(12)
Level 5	120(26)
Level 6	154(33)
Level 7 (Expert)	96(21)

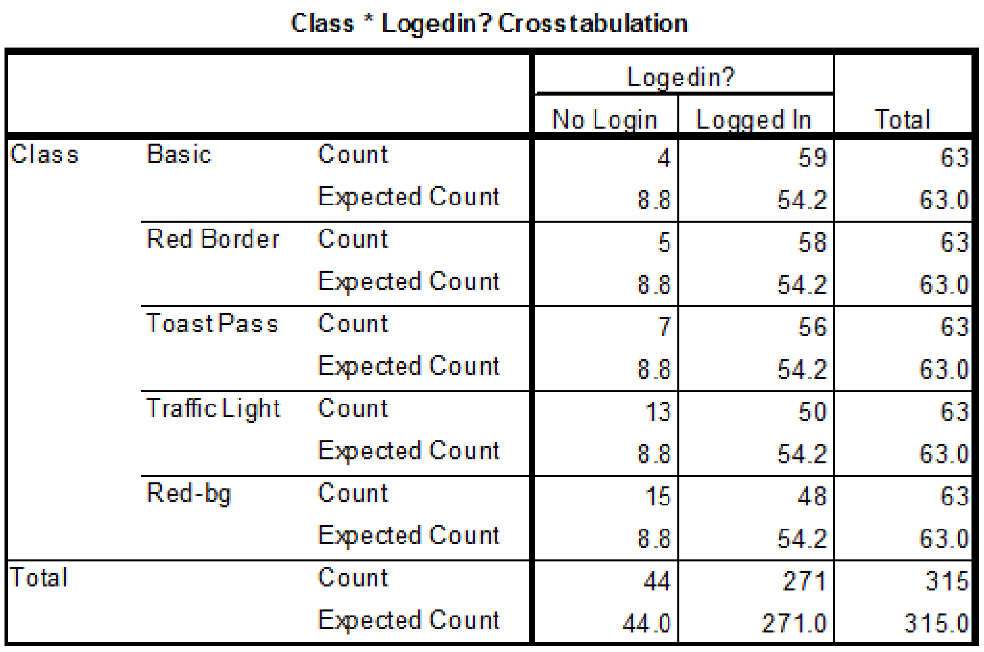
used third-party apps, and (32%) used them less often. All of this suggests that even though the majority of participants had an antivirus installed on their mobile devices, which means, security is very important to them. Yet, they have hight tendency to install apps on their mobile devices in a regular basis. Moreover, they frequently used third-party apps to access their online social accounts. This suggestion implies that the JavaScript injection attacks that we discussed earlier can have hight impact given the number of users who use third-party apps to access their online social accounts.

Table 19: Participants Mobile Phone Usage Pattern

Variable	n(%)
Antivirus is installed	
Yes	345(74)
No	120(26)
Apps Installation	
Level 1 (Never)	20(4)
Level 2	33(7)
Level 3	40(9)
Level 4	88(19)
Level 5	122(26)
Level 6	108(23)
Level 7 (Every time)	54(12)
Third-party Apps for social	
Level 1 (Never)	66(14)
Level 2	42(9)
Level 3	42(9)
Level 4	68(15)
Level 5	111(24)
Level 6	79(17)
Level 7 (Every time)	57(12)

5.2.7 The effect of the Security Cues on Participants Decisions

In our experiment, participants were asked to post a picture and a text on their Facebook accounts, which requires them to authenticate and authorize the application on their Facebook profiles or the study Facebook profile. Participants were given two options to do so, they could either use their own Facebook account or the study account (We created a temporary Facebook account for the study). The reason for creating this account is that our study was taken out of the Amazon Turk for violating their privacy rules since we require participants to login to Facebook using their accounts. We based our effectiveness on two factors: the first one by looking at the percentage of participants who refused to login at all, and second, by looking at



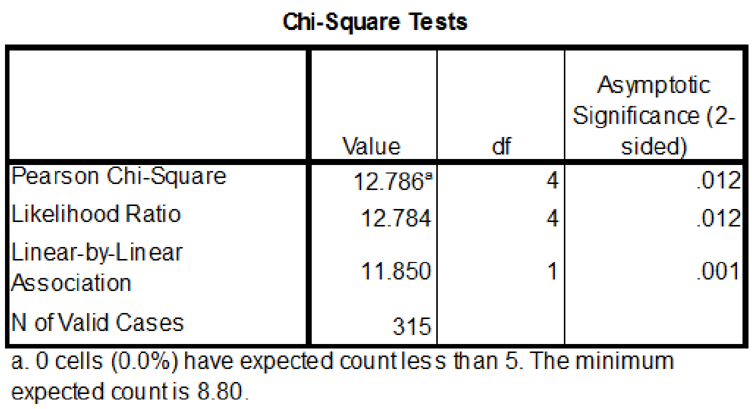
Class * Logedin? Crosstabulation

			Logedin?		
			No Login	Logged In	Total
Class	Basic	Count	4	59	63
		Expected Count	8.8	54.2	63.0
	Red Border	Count	5	58	63
		Expected Count	8.8	54.2	63.0
	ToastPass	Count	7	56	63
		Expected Count	8.8	54.2	63.0
	Traffic Light	Count	13	50	63
		Expected Count	8.8	54.2	63.0
	Red-bg	Count	15	48	63
		Expected Count	8.8	54.2	63.0
Total		Count	44	271	315
		Expected Count	44.0	271.0	315.0

Figure 34: SPSS Crosstabulation Results of the Logedin class.

the percentage of participants who used the study Facebook account instead of their own accounts. Figure 34 and Figure 35 shows the analysis results for the logedin class. The chi-square test statistic is 12.79 with 4 degree of freedom and an associated p <0.05. The null hypothesis is rejected, since p <0.05, and a conclusion is made that the presented cue design is associated with participants decision to login or not. Examining the pattern of numbers in Figure 34, it is noted that more participants decided not to login than expected when they are presented with the Red Background and Traffic Light cue designs and less participants decided not to login when they are presented with other security cue designs or nothing at all.

As per the relationship between the presented security cue design and account choice. Figure 36 and Figure 37 shows the analysis results for the account class. The null hypothesis is accepted, since p >0.05, and a conclusion is made that the presented cue design is not associated with the account choice.

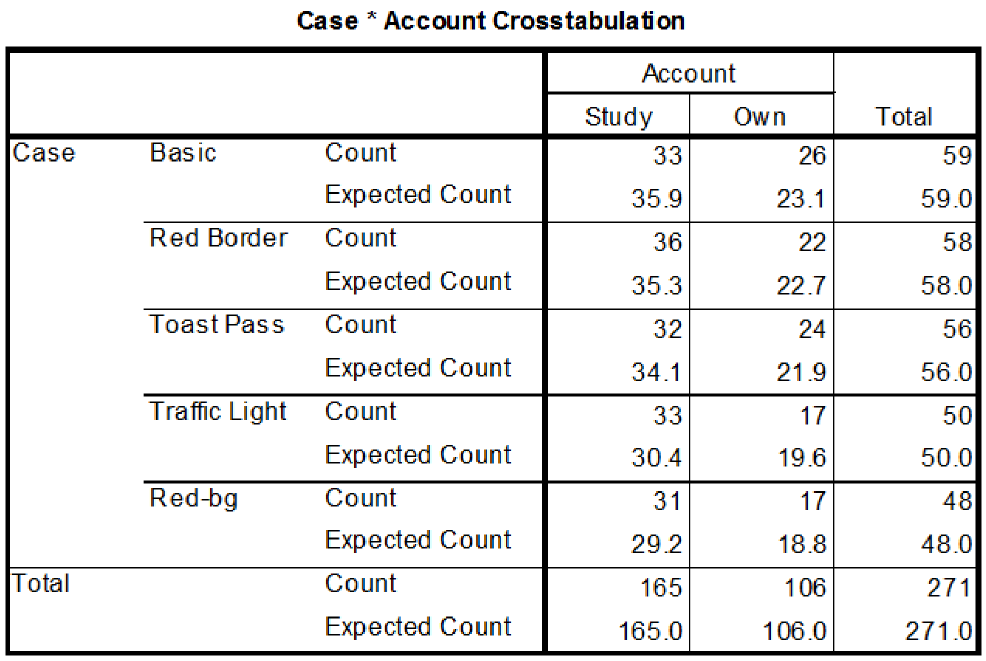


Chi-Square Tests

	Value	df	Asymptotic Significance (2-sided)
Pearson Chi-Square	12.786[a]	4	.012
Likelihood Ratio	12.784	4	.012
Linear-by-Linear Association	11.850	1	.001
N of Valid Cases	315		

a. 0 cells (0.0%) have expected count less than 5. The minimum expected count is 8.80.

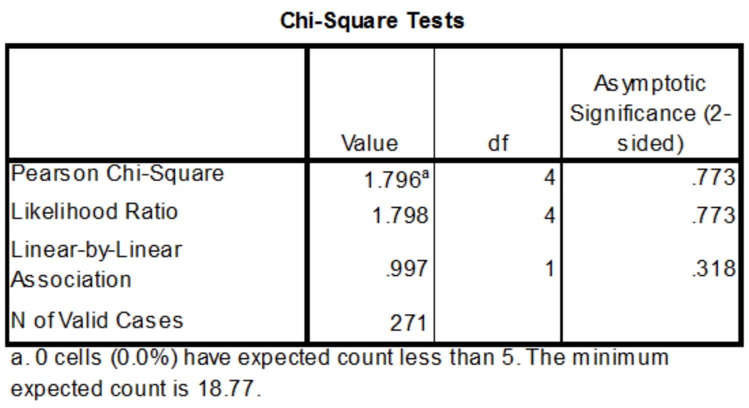
Figure 35: SPSS Chi-Square Results of the Logedin class.



Case * Account Crosstabulation

			Account		Total
			Study	Own	
Case	Basic	Count	33	26	59
		Expected Count	35.9	23.1	59.0
	Red Border	Count	36	22	58
		Expected Count	35.3	22.7	58.0
	Toast Pass	Count	32	24	56
		Expected Count	34.1	21.9	56.0
	Traffic Light	Count	33	17	50
		Expected Count	30.4	19.6	50.0
	Red-bg	Count	31	17	48
		Expected Count	29.2	18.8	48.0
Total		Count	165	106	271
		Expected Count	165.0	106.0	271.0

Figure 36: SPSS Crosstabulation Results of the Account class.



Chi-Square Tests

	Value	df	Asymptotic Significance (2-sided)
Pearson Chi-Square	1.796[a]	4	.773
Likelihood Ratio	1.798	4	.773
Linear-by-Linear Association	.997	1	.318
N of Valid Cases	271		

a. 0 cells (0.0%) have expected count less than 5. The minimum expected count is 18.77.

Figure 37: SPSS Chi-Square Results of the Account class.

Table 20: The reasons why participants chose not to login at all.

	Controlled	Red Border	Toast	Traffic	Red bg
I don't trust 3rd-party apps	1	1	4	4	4
I chose not to	1	2	1	3	2
I wanted to finish early	1	2	1	1	2
I saw something suspicious	0	0	1	3	5
Other	1	0	0	2	2
Total	4	5	7	13	15

Table 21: The reasons why participants chose the study Facebook account to login.

	Controlled	Red Border	Toast	Traffic	Red bg
I do not trust 3rd-party apps	5	14	7	4	5
I felt more comfortable	22	14	19	20	12
I chose to	4	4	3	6	1
I saw something suspicious	0	0	1	2	5
Other	2	4	2	1	2
Total	33	36	32	33	25

Digging deeper into why participants chose not to login at all, we asked them to specify the reasons. Table 20 shows participants answers with regard to the reasons that made them not to log in at all. The most important reason for us was the ”I saw something suspicious”, which shows the real effectiveness of the security design cues. As it appears, participants in the Red Border group did not notice the cue design at all. However, the Traffic Light and Red Background and Text designs remains the most noticeable ones. We also asked participants about why they chose the study Facebook account to login. Their answers are shown in Table 21. We are really very much interested in the fourth reason in that table and that is ”I saw something suspicious”. The Traffic Light and Red Background and Text cue designs made 7 participants not to use their own Facebook accounts, which proves their effectiveness over the other two designs. As far as the reasons participants have chosen their own Facebook account to login, another question was presented to them and their answers

Table 22: The reasons why participants chose their own Facebook account to login.

	Controlled	Red Border	Toast	Traffic	Red bg
I didn't see anything suspicious	7	3	3	2	1
I normally do	9	11	9	9	7
It was faster/easier	6	4	7	4	6
I chose to	4	4	5	2	2
Other	0	0	0	0	1
Total	26	22	24	17	17

Table 23: Chi-Square Test Results of the Visibility Variable - Frequencies Table.

Frequencies			
	Observed N	Expected N	Residual
Red bg and Text	99	63.0	36.0
Traffic Light	65	63.0	2.0
Toast Password	43	63.0	-20.0
Red Border	45	63.0	-18.0
Total	252		

are depicted in Table 22. As the table data shows, the main two reasons are simplicity and the fact that users are used to provide their Facebook credentials to third-party apps.

5.2.8 Comparing the Security Cues: Understandability and Noticeability

In this section we compare between the different security cue designs that were introduced into the authentication page to alert users of a suspicious activity that could affect the confidentiality of their credentials. Participants were presented with the four designs and asked to pick the most noticeable design and the most understandable design. As it appears from the Frequencies and Statistics Tables. It statistically significant as Pearson Chi-Square is 32.127 with 3 degree of freedom and an associated $p < 0.05$.

We studied the effect of the presented security design and participants' choices for the most visible and understandable security cue design. Our results showed that

Table 24: Chi-Square Test Results of the Visibility Variable - Statistics Table.

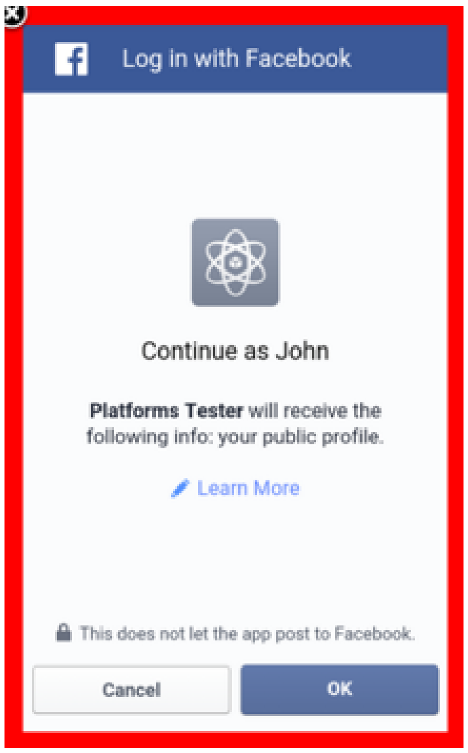
Test Statistics	
Chi-Square	32.127[a]
df	3
Asymp. Sig.	.000

Table 25: Chi-Square Test Results of the Understandable Variable - Frequencies Table.

Frequencies			
	Observed N	Expected N	Residual
Red bg and Text	43	63.0	-20.0
Traffic Light	126	63.0	63.0
Toast Password	40	63.0	-23.0
Red Border	43	63.0	-20.0
Total	252		

Table 26: Chi-Square Test Results of the Understandable Variable - Statistics Table.

Test Statistics	
Chi-Square	84.095[a]
df	3
Asymp. Sig.	.000


Log in with Facebook

Continue as John

Platforms Tester will receive the following info: your public profile.

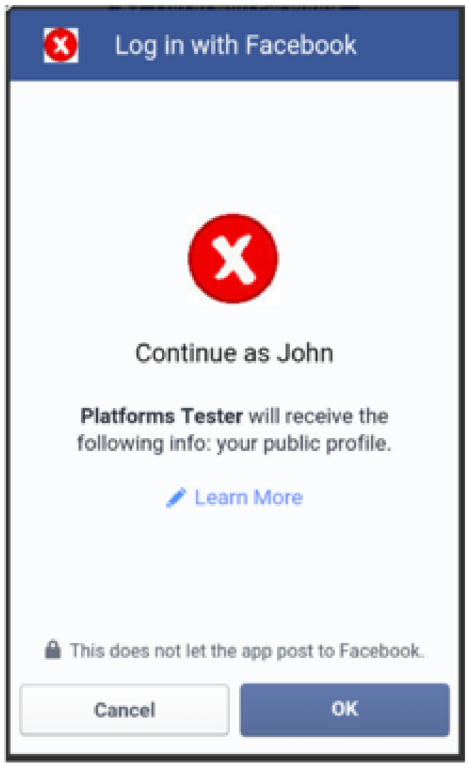
This does not let the app post to Facebook.

Figure 38: Design C displays a red border around the authorization page.

there is no significant relationship between the presented security cue design and participants' selections for the most visible and most understandable designs.

5.2.9 Security Cue Design for the Authorization Page

The authorization page normally displays the application name and the list of permissions it is requesting. Some malicious apps can modify this page to trick users into authorizing them, thus, an alert must be sent to the user if such malicious behavior is detected. Below are three different security cue designs that can be used to alert the users: Design A highlights part of the permissions and displays a warning message Figure 40, Design B uses the red X sign (Traffic Light) Figure 39, and Design C uses a red line surrounding the authorization page, Figure 38.


Log in with Facebook

Continue as John

Platforms Tester will receive the following info: your public profile.

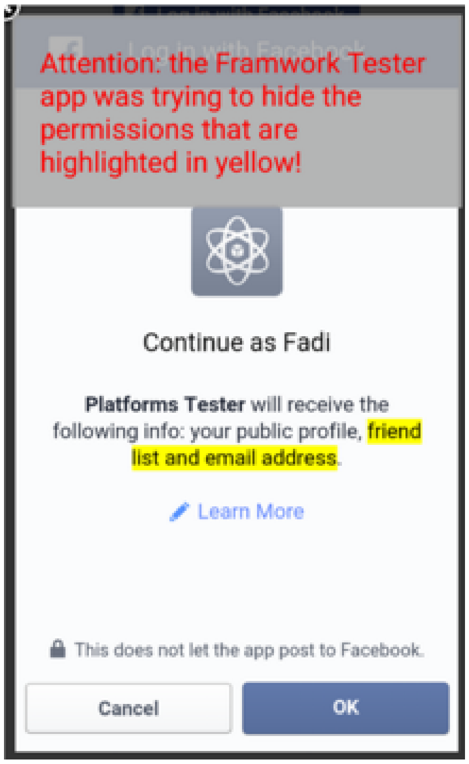
Learn More

This does not let the app post to Facebook.

Cancel

OK

Figure 39: Design B uses the red sign (Traffic Light).


Attention: the Framwork Tester app was trying to hide the permissions that are highlighted in yellow!

Continue as Fadi

Platforms Tester will receive the following info: your public profile, friend list and email address.

Learn More

This does not let the app post to Facebook.

Cancel

OK

Figure 40: Design A highlights parts of the permissions and display a toast/warning message).

5.2.10 Authorization Page - Survey Questions

The survey questions that we used are identical to the ones we used in the authentication part with few minor changes:

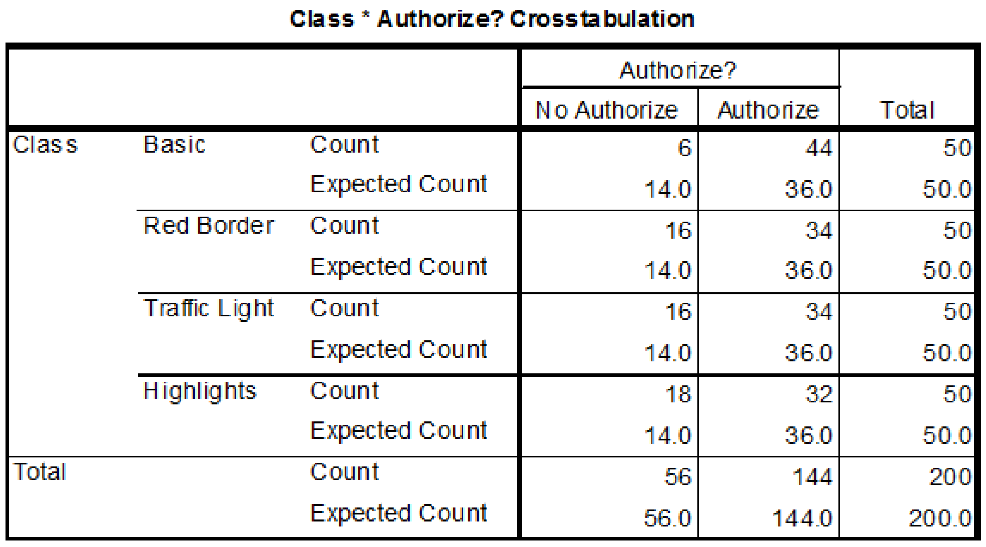
- Please describe the suspicious elements that you saw on the authorization (permissions) page?
- Which one of these designs did you see when you used the Platfroms Tester App?
  - Design A.
  - Design B.
  - Design C.
  - None.
- Which one of these designs is the most visible (noticeable)?
  - Design A.
  - Design B.
  - Design C.
- Which one of these designs is the most understandable?
  - Design A.
  - Design B.
  - Design C.

In the authorization page, the integrity of the displayed information, the permission list in particular, is of our top most concern. Previous studies have shown that Android users pay less attention to the requested permissions when installing apps on their mobile devices. In this study, we are experimenting with different security cue designs that could make users pay more attention especially when the requested permission are being manipulated. In comparing between the different security cue designs, we investigated the relationship between the presented security cue design and participants decisons to authorize the application on their Facebook/Study accounts or not. Figure 41 and Figure 42 shows the analysis results for the authorize class. The chi-square test statistic is 8.73 with 3 degree of freedom and an associated $p < 0.05$. The null hypothesis is rejected, since $p < 0.05$, and a conclusion is made that the presented cue design is associated with participants decision to authorize or not. Examining the pattern of numbers in Figure 41, it is noted that more participants decided not to authorize than expected when they were presented with the Highlights, Red Border, and Traffic Light cue designs and less participants decided not to authorize when they were presented with no security cue design.

As far as why participants chose to authorize the application, Table 27 summarizes participants' answers. On the other hand, Table 28 gives a summary of the reasons that made participants authorize the application.

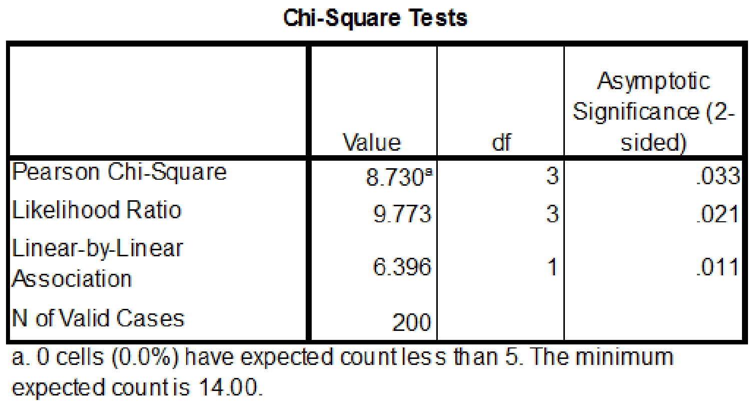
5.2.11 Comparing the Security Cues - Authorization Page: Understandability and Noticeability

In this section we compare between the different security cue designs that were introduced into the authorization page to alert users of any suspicious activity that


Class * Authorize? Crosstabulation

			Authorize?		Total
			No Authorize	Authorize	
Class	Basic	Count	6	44	50
		Expected Count	14.0	36.0	50.0
	Red Border	Count	16	34	50
		Expected Count	14.0	36.0	50.0
	Traffic Light	Count	16	34	50
		Expected Count	14.0	36.0	50.0
	Highlights	Count	18	32	50
		Expected Count	14.0	36.0	50.0
Total		Count	56	144	200
		Expected Count	56.0	144.0	200.0

Figure 41: SPSS Crosstabulation Results of the Authorize class.


Chi-Square Tests

	Value	df	Asymptotic Significance (2-sided)
Pearson Chi-Square	8.730[a]	3	.033
Likelihood Ratio	9.773	3	.021
Linear-by-Linear Association	6.396	1	.011
N of Valid Cases	200		

a. 0 cells (0.0%) have expected count less than 5. The minimum expected count is 14.00.

Figure 42: SPSS Chi-Square Results of the Authorize class.

Table 27: The reasons why participants chose to authorize the application on their/Study Facebook account.

	Controlled	Red Border	Traffic Light	Highlights	Sum
Nothing suspicious	12	10	12	15	49
I normally do	11	11	12	10	44
I had to	18	11	8	5	42
Other	3	2	2	2	9
Sum	44	34	34	32	144

Table 28: The reasons why participants chose to authorize the application on their/Study Facebook account.

	Controlled	Red Border	Traffic Light	Highlights	Sum
I do not trust	2	6	5	7	20
I chose not to	2	3	2	2	9
To finish early	1	5	3	0	9
I saw something	0	0	5	7	12
Other	1	2	1	2	6
Sum	6	16	16	18	56

Table 29: (Authorization) Chi-Square Test Results of the Understandable Variable - Frequencies Table.

Frequencies			
	Observed N	Expected N	Residual
Red Border	51	50.0	1.0
Traffic Lights	37	50.0	-13.0
Highlights	62	50.0	12.0
Total	252		

could affect the integrity of the presented information, the permissions list. Participants were presented with the three designs and asked to pick the most noticeable design and the most understandable design of these three. As it appears from the Frequencies and Statistics Tables of the understandable class: Table 29 and Table 30. It statistically significant as Pearson Chi-Square is 6.280 with 2 degree of freedom and an associated p <0.05. The highlights design was chosen by more participants is the most understandable design than any the other two designs. As far as the visibility class, the Frequencies and Statistics Tables of the visibility class: Table 31 and Table 32 shows that there is no significant difference between the three designs in terms of visibility. However, the Highlights design scores slightly higher than the other two.

We studied the effect of the presented security design and participants‘ choices for

Table 30: Authorization) Chi-Square Test Results of the Understandable Variable - Statistics Table.

Test Statistics	
Chi-Square	6.280[a]
df	2
Asymp. Sig.	.043

Table 31: (Authorization) Chi-Square Test Results of the Visibility Variable - Frequencies Table.

Frequencies			
	Observed N	Expected N	Residual
Red Border	44	50.0	-6.0
Traffic Lights	48	50.0	-2.0
Highlights	58	50.0	8.0
Total	252		

Table 32: (Authorization) Chi-Square Test Results of the Visibility Variable - Statistics Table.

Test Statistics	
Chi-Square	2.080[a]
df	2
Asymp. Sig.	.353

the most visible and most understandable security cue design. Our results showed that there is no significant relationship between the two.

5.2.12 Users' Privacy Assumptions: Web Applications Vs. Mobile Applications

The OAuth is used to authorize both web applications and mobile applications. In both cases, users have to authenticate themselves and then authorize these applications on their online profiles. In this dissertation, we showed the crucial difference between the two applications in terms of maintaining the integrity of the authorization pages and assuring the confidentiality of the authentication pages' data. However, the difference might not be well realized by the end user. Thus, we asked participants to answer 12 questions that capture their privacy assumptions about both applications. Six questions are about mobile applications and the other six are about web applications. In this section, we compare between the two assumptions by analyzing participants' responses to these questions by comparing 6 pairs. The users have to rely on their experiences in completing the first part of the study and their expediences in using mobile applications to answer the mobile applications' related questions. Moreover, participants were given a web application scenario and asked them to answer the second set of questions based on it and based on their experiences with using web applications.

Question 1 (Mobile) : When entering your username and password in Page 1 (authentication) you are sharing your username and password with Facebook only.

Question 1 (Web) : When entering your username and password in Page 2 (authentication) you are sharing your username and password with Facebook only.

Goodman and Kruskal's gamma was run to determine the association between users'

Table 33: The descriptive statistics for participants' responses to Question 1.

Descriptive Statistics.					
	N	Minimum	Maximum	Mean	Std. Deviation
Fbonly	465	1.0	7.0	4.83	1.81
Fbonlyweb	465	1.0	7.0	4.89	1.71

privacy assumptions about web applications and mobile applications amongst 465 participants. There was a strong, positive correlation between users' privacy assumptions about web applications and mobile applications, which was statistically significant (G = .662, p ¡ .0005) as it appears from Symmetric Measures values shown in Table 34 respectively. Table 33 shows that the average of participants' answers to the web applications is higher than the mobile applications, which can be interpreted as that users have more trust in web applications than mobile applications.

Question 2 (Mobile) : When entering your username and password in Page 1 (authentication) you are sharing your username and password with Facebook and the third-party mobile app.

Question 2 (Web) : When entering your username and password in Page 2 (authentication) you are sharing your username and password with Facebook and the Doodle website.

The analysis we applied on Question 1 data was also applied on Question 2, we got very similar results.

Question 3 (Mobile): I think that mobile apps are able to access the contents Page 1(authentication) and the user entered data (username and/or password).

Question 3 (Web) : I think that the Doodle website is able to access the contents of Page 2 (authentication) and the user entered data such as the username and pass-

Table 34: The Symmetric measures for participants' responses to Question 1.

Symmetric Measures				
	Value	Asymptotic Standardized Error [a]	Approximate T [b]	Approximate Significance
Ordinal by Ordinal Gamma	.662	1.0	7.0	.000
N of Valid Cases	465			

Table 35: The descriptive statistics for participants' responses to Question 3.

Descriptive Statistics.					
	N	Minimum	Maximum	Mean	Std. Deviation
Access Cred Mobile	465	1.0	7.0	4.98	1.63
Access Cred Web	465	1.0	7.0	4.78	1.77

word.

Goodman and Kruskal's gamma was run to determine the association between users' assumptions with regard to whether web applications and mobile applications can access the presented permissions amongst 465 participants. There was a strong, positive correlation between users' privacy assumptions about web applications and mobile applications, which was statistically significant (G = .591, p ¡ .0005) as it appears from Symmetric Measures values shown in Table 36 respectively. Table 35 shows that the average of participants' answers to the mobile applications is higher than the web applications, which can be interpreted as that users have less trust in mobile applications in comparison to applications.

Question 4 (Mobile) : I usually look carefully on the list of permissions that third-party mobile applications are requesting (Page 2) on my social accounts (e.g. Facebook).

Question 4 (Web) : I usually look carefully on the list of permissions that a website such as Doodle is requesting on my social accounts (Page 3).

Table 36: The Symmetric measures for participants' responses to Question 3.

Symmetric Measures				
	Value	Asymptotic Standardized Error [a]	Approximate T [b]	Approximate Significance
Ordinal by Ordinal Gamma	.591	.038	14.804	.000
N of Valid Cases	465			

Table 37: The descriptive statistics for participants' responses to Question 4.

Descriptive Statistics.					
	N	Minimum	Maximum	Mean	Std. Deviation
Cred Careful Mobile	465	1.0	7.0	5.17	1.55
Cred Careful Web	465	1.0	7.0	5.25	1.47

Goodman and Kruskal's gamma was run to determine the association between users' habits in looking at the requested permissions by web applications and by mobile applications amongst 465 participants. There was a strong, positive correlation between users' privacy assumptions about web applications and mobile applications, which was statistically significant (G = .697, p ¡ .0005) as it appears from Symmetric Measures values shown in Table 38 respectively. Table 37 shows that the average of participants' answers to the mobile applications is less than the web applications, which can be interpreted as that users are less careful to look at the requested permissions by the mobile applications in comparison to the web applications. This conclusion goes very well with the analysis of Question 6 in that since the users believe that the mobile applications are capable of modifying the requested permissions, they are less careful in looking at them before authorizing the app.

Question 5 (Mobile) : I feel not safe granting third-party mobile applications any privileges over my social accounts (e.g. Facebook).

Question 5 (Web) : I feel not safe granting websites such as Doodle any privilege over

Table 38: The Symmetric measures for participants' responses to Question 4.

Symmetric Measures				
	Value	Asymptotic Standardized Error [a]	Approximate T [b]	Approximate Significance
Ordinal by Ordinal Gamma	.697	.032	19.99	.000
N of Valid Cases	465			

Table 39: The descriptive statistics for participants' responses to Question 5.

Descriptive Statistics.					
	N	Minimum	Maximum	Mean	Std. Deviation
Not safe Mobile	465	1.0	7.0	5.14	1.55
Not safe Web	465	1.0	7.0	4.91	1.61

my social accounts (e.g. Facebook).

Analyzing participants' responses to Question 5 confirms the previous two occlusions as it shows that the end users do not feel safe granting third-party mobile applications any privileges over their social accounts. Table 39 and Table 40 shows the results of the analysis for both the web and mobile applications.

Question 6 (Web) : I think that websites such as Doodle are able to access the contents of Page 3 (authorization). (for instance, it can manipulate the list of permissions).

Question 6 (Mobile) : I think that mobile apps are able to access the contents of Page 2 (authorization).

Table 40: The Symmetric measures for participants' responses to Question 5.

Symmetric Measures				
	Value	Asymptotic Standardized Error [a]	Approximate T [b]	Approximate Significance
Ordinal by Ordinal Gamma	.647	.035	16.92	.000
N of Valid Cases	465			

Table 41: The descriptive statistics for participants' responses to Question 6.

Descriptive Statistics.					
	N	Minimum	Maximum	Mean	Std. Deviation
Access perm Mobile	465	1.0	7.0	5.11	1.48
Access perm Web	465	1.0	7.0	4.99	1.62

Table 42: The Symmetric measures for participants' responses to Question 6.

Symmetric Measures				
	Value	Asymptotic Standardized Error [a]	Approximate T [b]	Approximate Significance
Ordinal by Ordinal Gamma	.627	.037	15.63	.000
N of Valid Cases	465			

Similarly, analyzing participants' responses to Question 6 shows that end users believe that third-party mobile applications can access and modify the presented permissions. Table 42 and Table 41 shows the results of the analysis for both the web and mobile applications.

CHAPTER 6: KEYLOGGER THREAT IN ANDROID

6.1 Keylogger Threat

Unlike the majority of mobile operating systems, Android was the first mobile operating system to allow users to install third-party keyboards. To be more precise, Android started supporting the installation of third-party keyboards since version 1.5 (Known for its code name, Cupcake [4]). Users of other mobile operating systems, such as iOS [77], have had a long standing criticism of iOS for not allowing the installation of third-party keyboards. However, with the introduction of iOS 8 –the biggest iOS release ever according to Apple– third-party keyboards were finally allowed on both iPhone and iPad. Third-party keyboards can be seen as an opportunity to enjoy new features, security experts on the other hand see them as a potential threat to users‘ privacy. The moment a user installs and activates a third-party keyboard in his/her mobile device, all of his/her keystrokes might be leaked or misused. This opens the door for malicious developers to create keyloggers for the purpose of spying and/or fishing for users' sensitive data. Users may unknowingly install keyloggers off the online markets or may use a keylogger that a malicious user with physical access has installed on their devices. A malicious developer may build a keylogger from scratch or use an existing keyboard. In both scenarios, certain capabilities (permissions) must be available to these keyboards such as the Internet capability. Thus, it is the responsibility of mobile operating systems, third-party keyboard providers, and mobile users to ensure the safety of these keyboards.

Android and iOS have been seen to be completely different in terms of their supported security features. Both Android and iOS use traditional access control, isolation, permissions-based access control, and limited hardware access. However, they differ in other security aspects such as application security testing. Application security testing refers to the level of verification on third-party applications before been admitted into the market or installed on users‘ phones. If you look at the two platforms in regard to verifying third-party keyboards, you will find that iOS poses more limitations on keyboards than does Android operating systems. Thus, Android third-party keyboard providers have to share the responsibility of protecting users data on their servers. They also need to clearly state their terms and conditions and allow users to choose to enable the advanced features, especially those that require sensitive permissions, like INTERNTE. Last, the users must be aware of the fact that installing and activating third-party keyboards can compromise their privacy.

In this paper, we are conducting a user study to collect more insightful information in regard to third-party keyboards and their potential risks. For the purpose of this study we created an app that scans the Android devices for installed third-party keyboards to give a security assessment based on the permissions they possess and other security configurations. The app also contains a survey about users‘ usage behavior and their understanding to security configurations. Moreover, the app collect information off the Android devices to compare with users‘ survey answers. The collected data shows the tendency of Android users to install third-party keyboards, moreover, the majority of these keyboards possess sensitive permissions that make them capable of causing harm to user data. Previous researches have pointed out the danger of keylogging threat on Android users [13, 17, 54].

6.1.1 Privacy Risks

A malicious third-party keyboard once activated can leak any data the end user enters into his or her device. The data can be text messages, emails, credentials, phone numbers, social security numbers, credit card numbers, etc. Any of these data types can be misused to achieve damage, profit or both [17]. For example, an attacker may use stolen text messages to blackmail their owners or use stolen social security numbers to commit a fraud. Thus, the severity of any keylogger attack depends largely on the device vulnerability level and user usage behavior. For instance, a keylogger attack against a rooted Android device that is used by an end user to do banking and economical activities may result in a severe damage or loss. For this reason, our user study is focused towards capturing and collecting users usage behaviors and possible vulnerabilities.

6.1.2 Risky IME

There are many ways to build and distribute malicious third-party keyboards in Android. One of which is to convert a benign keyboard into a keylogger trojan and then make it available on non-official App Stores. The conversion can be done by changing the Java source code or Smali source code and then compile and resign the APK . Android apps are built using Java which is compiled to byte code to run on the Dalvik virtual machine. Nowadays, there exist many tools to reverse engineer Android APKs, modifying their source code, compile and sign them back: *apktool*, *keytool*, and *APK multitool*. An Android developer had used these tools to demonstrate the feasibility of converting the *SwiftKey* keyboard into a keylogger that send all keylogs to a particular server [15]. The other way is building a malicious keyboard from scratch. Android online developers guide provide tutorials on building third-party keyboards [38]. Moreover, Android provide code samples for developers to download and use in their projects, for instance, an Android code sample for SoftKeyboard can be find here[39]. The malicious keyboards that are developed using the later two methods can be then placed into official and non-official online app stores. A malicious third-party keyboard attempting to steal user‘s entered would either store it for on the device at first then send it to a remote server or send it directly to a remote server [54, 13].

6.1.3 Vulnerable Device

The vulnerability of a device is related to the keylogging‘s attack vector and adversary model. A keylogger can be built from scratch and distributed over online markets or it can be a modified version of a benign keyboard that is distributed over non-official markets. The victim may then install any of these keyboards into his or her device unknowingly of the suspicious behavior. Attackers may also have a physical access on the victim device that give him or her the ability to install and activate their malicious keyboards. According to the previous definitions, a vulnerable device has any of the following characteristics: rooted, screen unlock is not activated, and allows installation from unknown sources. Our app extracts these information from participants‘ devices and gives a safety summary score accordingly.

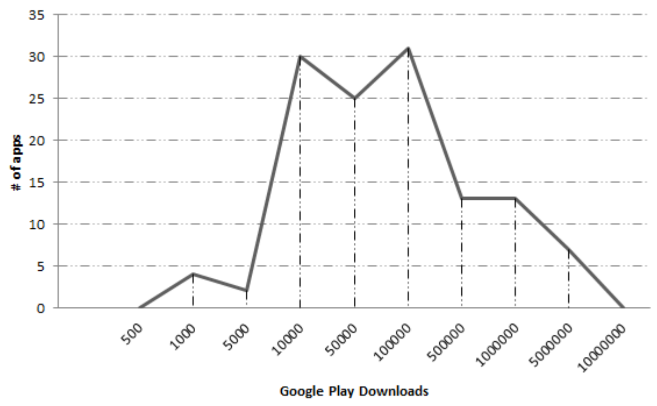


Figure 43: The download frequency for the collected third-party keyboards

6.2 Empirical Study

We present in this paper the first empirical study on Android third-party keyboards phenomenon, giving many details about these keyboards. In this paper our data analysis is centralized around two major points. The first is concerning the number and type of permissions requested by the collected keyboard applications set, which tells us the vulnerability of the keyboard application. The second is the number of users downloading these applications and their ratings, which tells us the scale of an attack. We conducted an empirical study on a set of 125 keyboards apps collected from Google Play Market. We used the apktool to get their *AnadroidManifest* files. We then parse these files for the permission requests. Moreover, we use the data; app installation count and ratings, available on the Android Market information page.

6.2.1 Permissions Analysis

Among the studied 125 keyboard applications, only 15% requested zero permissions, the remaining 85% requested one or more permissions upon installation, see Figure 47. The number of permissions requested ranges from 1 to 16 with few ex-

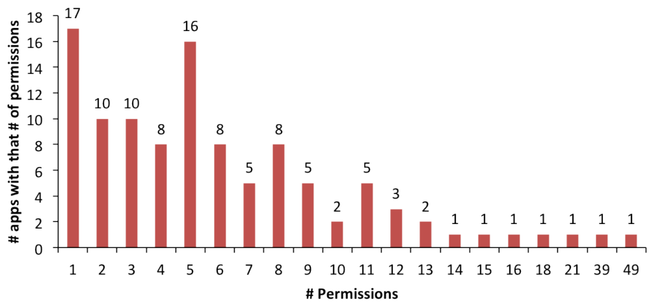


Figure 44: The frequency for the number of permissions requested by collected apps

ceptions. Figure 46 displays the frequency for each number of permission, 61% of these applications requested more than two and less than eight permissions. The highest frequency is seventeen for one permission. As an extreme case, we found a keyboard application that requested forty nine permissions. The application has a rating of 4.6 out of 5 and number of downloads were between five hundred thousands and one million. For the type of permissions requested, in Figure 45, we show the most requested permissions overall keyboard apps. The permission android.permission.VIBRATE was the top most requested permission, 77.60% of the apps requested this permission.

6.2.2 Popularity

Number of downloads and users' ratings are two measures for the popularity of any Android application. Thus, as these two measures go higher, attackers get attracted to use like these applications to host their malicious code. Figure 46 shows the ratings distribution among all applications, 62% of the applications have rating range between 4.1 and 4.6. For the number of downloads, in Figure 43 we show the downloads distribution. From the figure, there are thirty applications with the range of fifty thousands and one hundred thousands downloads. More than 50% of applications

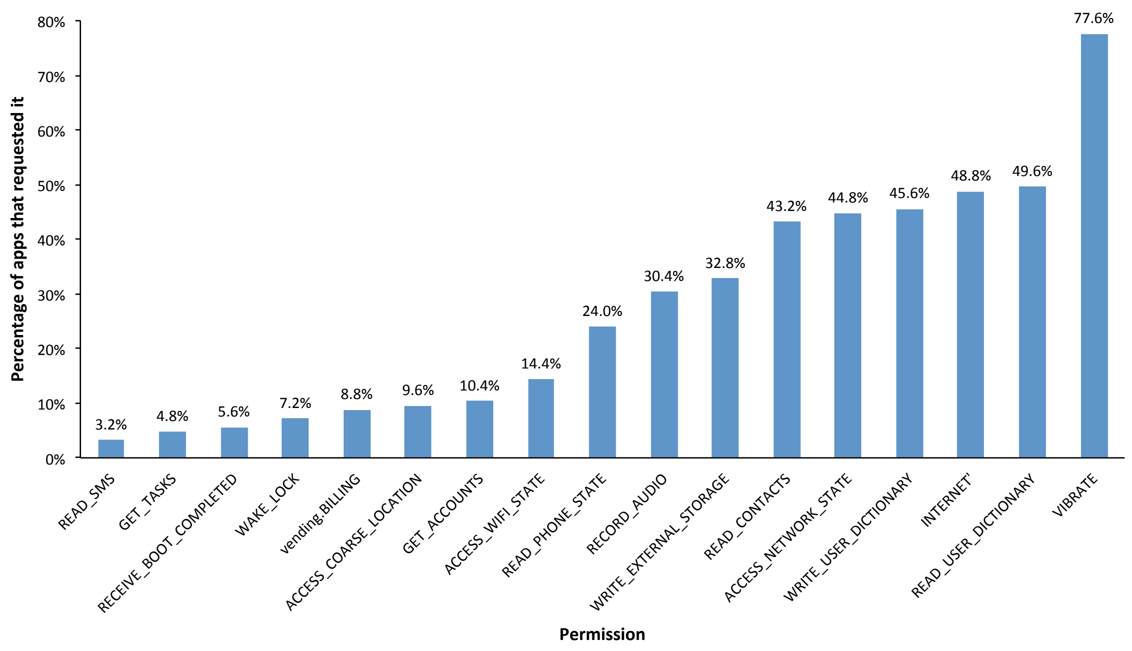


Figure 45: The 17 most requested permissions and the percentage of third-party keyboards that request them

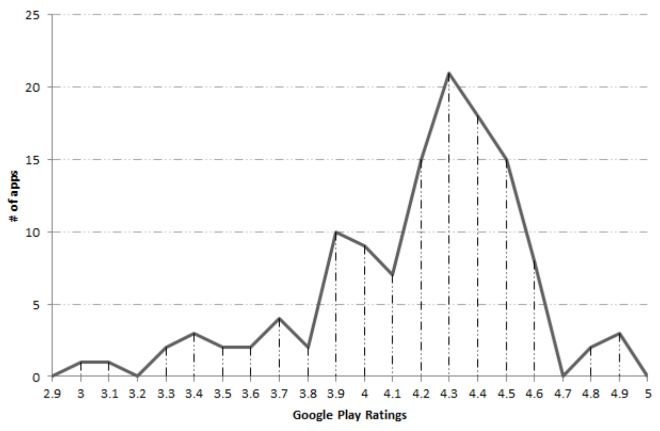


Figure 46: The rating frequency for the collected third-party keyboards

have download rates between fifty thousands and five millions.

Keyboard apps fall under the "Productivity" category according to Google categorization [29]. AppBrain [7] provides statistics for all Android app categories, follows are some information particularly given for the "Productivity" category:

1. The total number of apps in this category is 21154 apps

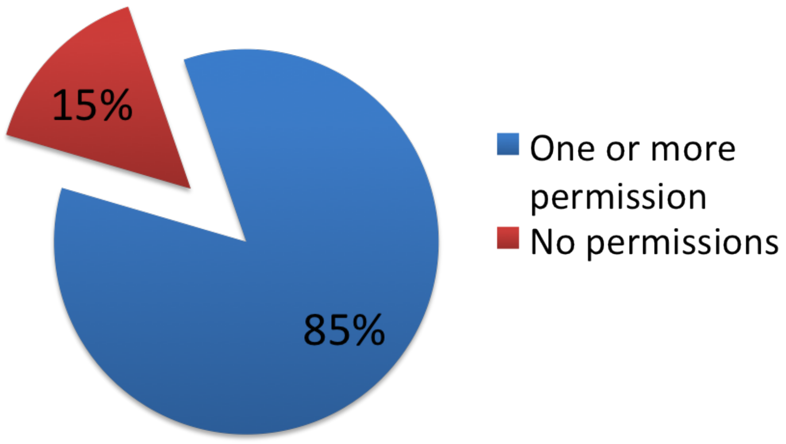


Figure 47: Among the studied set, 15% requested 0 permission, 85% requested one or more permission

2. The average star rating is 4.0 out of 5.0

3. The number of apps in the category that have more than 50,000 downloads is 1700 apps which is about 8% of the apps in the "productivity" category.

Based on our results and AppBrain statistics, we conclude that most third-party keyboards request permissions that would make them vulnerable to some serious attacks: 48.8% of these apps are vulnerable to type A attack, 42% to type A+, 29% to type B attack, and 24% to type B+. The number of downloads as well as users' numerical rating is considerably higher than the average compared to the apps of their category.

6.3 User Study

In this section we show the results of our Android user study. We asked participants to download our app on their Android devices to answer the embedded survey questions. The questions are centered around their Android devices usage behavior including installing third-party applications, security configurations, and accessing

Table 43: Participants Demography

Variable	n(%)
Gender	
Male	83(90)
Female	8(9)
Age	
18-24	26(29)
25-34	51(65)
35 and Above	14(15)
Education	
2 years of college	15(17)
4 years of college	42(46)
More than 4 years of college	34(37)
Familiarity with Android	
Less than 1 year	18(20)
1-4	60(65)
Greater than 4	14(15)
Purpose of Installation	
Functional	54 (59)
Luxury	8(9)
Found it installed	9(10)
No particular reason	20(22)
Choosing a Keyboard	
Random	16(17)
Cheaper	13(14)
Fewer Permission	30(33)
Vendor	33(36)

sensitive information.

6.3.1 Recruiting Participants

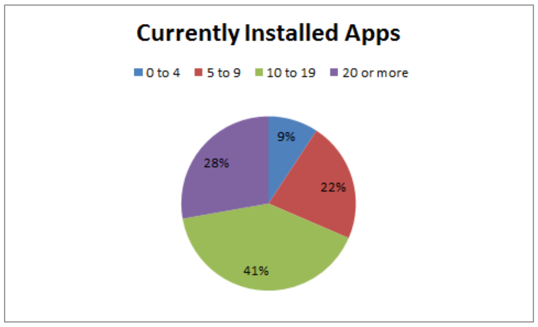
Recruiting participants was done by Amazon Turk, sending emails, sending SMS, and social media. For the Amazon Turk, we paid each participant $0.2 for completing the embedded survey and $0.2 for completing the post survey.

6.3.2 Number of Participants

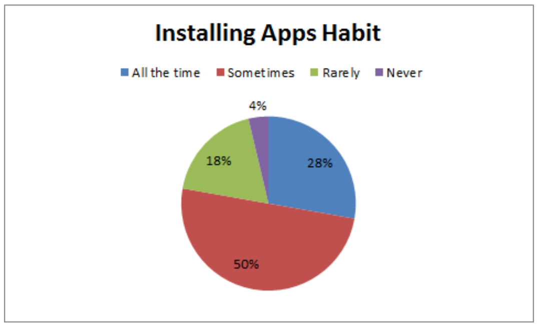
We had 92 (*16 questions) participants completing the embedded survey, 52 of them agreed to complete the post survey.

6.3.3 Embedded Survey

Participants upon running the AntiLoggers app for the first time are prompted to agree to the terms and conditions of the study. Following that, participants are prompted to answer the survey questions pertaining to their demography, familiarity with Android, and more.



(a) The number of Apps that are currently installed on participants‘ Android devices.



(b) The frequency with which participants get to install Apps on their devices.

Figure 48: The number and frequency of installing third-party application by the participants.

6.4 Analysis

In this section we show the results of our Android user study.

6.4.1 Participants Demography

The distribution of participants’ demography is as follows: of the 93 participants who downloaded the application to participate in the study, the majority (56%) were ages 25-34, and 29% were between 18-24. The remaining 15% fell between the ages of 35-74. Nearly half the participants (47%) had completed 4 years of college, while 37% had completed more than 4 years of college, and the remaining 17% had completed at least 2 years of college. With regards to gender, 91% (85 participants) were male while 9% (8 participants) were female. Lastly, the majority (60%) of all participants possessed an Android device running API 19 (codename ”Kit Kat”). Another 31% had an Android device running API 16, 17, or 18, (”Jelly Bean”), while 8% had Android API 15 (”Ice Cream Sandwich”), and 1% ran the latest API 21 (”Lollipop”).

Given that users’ general familiarity with the Android system is potentially relevant to users‘ awareness and understanding of the keylogging threat in Android, participants were evaluated on this familiarity, as represented by the charts in Table 43. For example, 48% of the participants had 1 to 3 years experience dealing with Android devices, and another 32% had 3 or more years of experience. Additionally, 41% of participants had 10-19 apps installed on their device while another 28% had 20 or more, and half of all participants installed new apps on a semi-regular basis, see Figure 48(a) and 48(b). All of this suggests that users are very familiar with downloading and using Android apps. Furthermore, it may actually be this comfort with installing third-party apps that increases the threat of exploit by malicious keylogging apps.

Figure 49: The security configurations for participants’ Android devices.

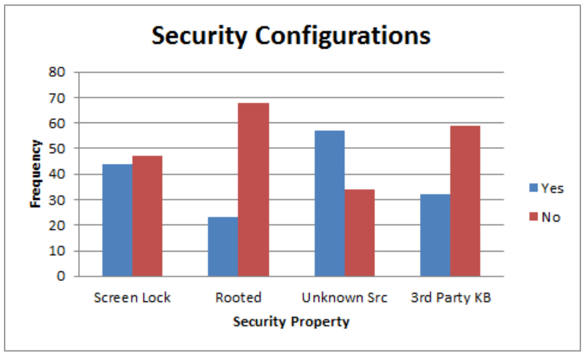


Figure 50: Measuring participants' security awareness by comparing device real configurations with their answers (Rooting).

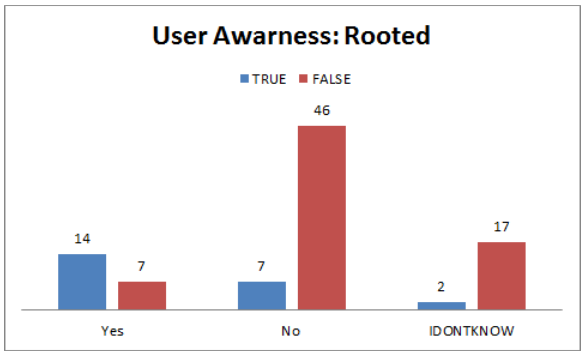
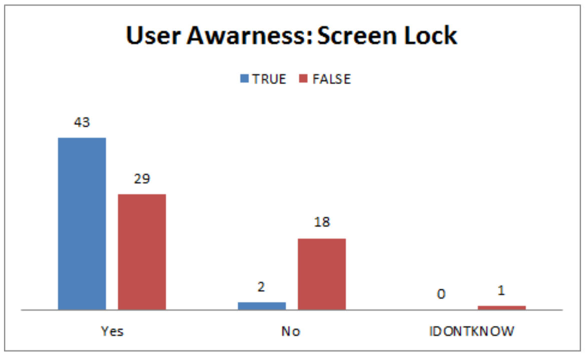


Figure 51: Measuring participants' security awareness by comparing device real configurations with their answers (Screen Lock).



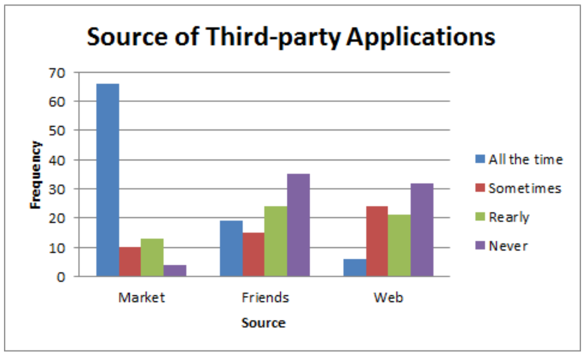
6.4.2 Automatically Collected Data

The AntiLoggers app was designed to collect a certain set of security configurations of participants' devices automatically, without user intervention. Based on these pieces of information, we are able to establish some basis of awareness that users had regarding the security of their device and the privacy of the information contained there within. These configurations included whether a screen lock had been enabled in the device, whether the device was rooted, whether the installation of apps from unknown sources (markets other than the main Android Play Store) was enabled, and whether any third-party keyboard apps existed on the device. The aggregate results of these automatically collected data can be found in Figure 49.

The most polarized of groups in the different security configuration levels are those individuals who claimed to have a rooted device compared to those who claimed do not, the latter of which is a significantly larger group. We believe this is to be expected, however, as Android devices are not rooted by default, when obtained from an authorized, first-party seller. Rather, a user has to actively root their device, and it is likely that only advanced Android users would take this step. Similarly, allowing apps to be installed from unknown sources is something that is not enabled by default in Android, for security purposes. Among our participants, however, there were actually more participants that claimed to have this enabled than disabled. Concerning third-party keyboard apps specifically, fewer participants in the study claimed to have these installed than not installed. Lastly, the screen lock security feature was almost as likely to be enabled in participants' devices as disabled.

It is interesting to note here that in addition to automatically collecting information on these security configurations, we included questions in the app's embedded survey that directly addressed them. In this way, we could compare what participants claimed to be enabled or configured in their device with what we programmatically found to be configured, in order to measure user awareness. This comparison revealed notable differences in user awareness, particularly concerning rooted devices and screen locks. With users' awareness of the root state of their devices, we can see in Figure 50 that 86% of the participants that claimed not to have a rooted device (46 out of 53) were actually incorrect, and one-third of those who claimed to have a rooted device (7 out of 21) were also incorrect. Similarly, in Figure 51, we see that 29 out of 72, or 40% of those who claimed to have a screen lock enabled were incorrect, while 16 out of 18 participants (88%) who claimed not to have a screen lock enabled actually did. While it is reasonable to consider a typical user's lack of understanding of what it means to "root" your Android device may have contributed to inaccurately responding to this question, it is strange that the same discrepancy occurred when addressing the screen lock.

Figure 52: The distribution of different Apps resources.



6.4.3 Survey Data

Within the AntiLoggers app, participants had to answer the embedded survey questions before accessing the core functionality of the app. The demographic information and the questions concerning rooted devices and screen locks that were referenced earlier were among these embedded survey questions. In addition, there were questions addressing the purpose for installing third-party keyboards, and motivation for choosing between different variations of keyboards. There were also questions pertaining to typical uses of the mobile device as well as sources for obtaining mobile apps. Together, the responses to these survey questions provided useful insight into the typical behavior of users regarding third-party keyboards and mobile apps in general, which then paints a picture of what level of vulnerability users are generally susceptible to through keylogging threats.

Analyzing the questions that are related to third-party keyboard apps first, we found that the majority of participants (59%) chooses to install third-party keyboards for the purpose of function over anything else. Furthermore, if choosing between two keyboard apps, 36% choose based on the vendor of the app, following closely by 33% indicating they would choose based on the app with fewer permissions. These are both reflected in Table 43. From this we gather that the majority of users install third-party keyboards out of perceived necessity; it is likely that the standard keyboard does not easily offer the support required, perhaps related to other language keyboards. Nonetheless, it is promising to see that the number of Android permissions requested does play a prominent role in the decision-making process, especially given that research shows that malicious keylogging applications often request a similar set of permissions. Figure 54 reveals what participants indicated they spent the most time using their Android devices for: browsing the web, checking and sending emails, and exploring social media. Where usage was more occasional than frequent was in banking and gaming. Furthermore, when it came to the source of obtaining third-party applications in general, the default market–Android Play Store–stood far above the rest, with more than 60 participants agreeing this is their more frequent resource. The other two options, that is from friends and from the web, saw more participants responding "Never" than any other frequency. Finally, we asked participants to complete a post survey and only 54 of them have agreed to do it. Table 44 provides the synopsis of the results of our post survey. Overall, the participants seemed to benefit from our app in learning about keyloggers and enhance their security awareness.

Figure 53: Participants responses to the security questions.

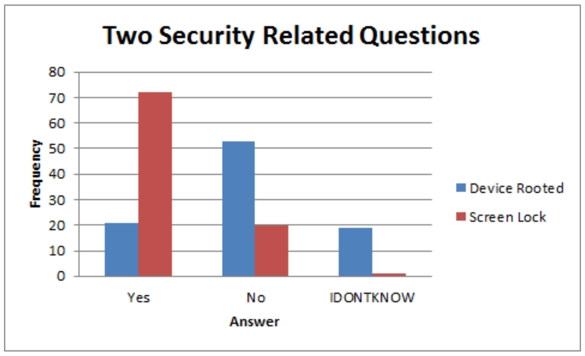


Figure 54: Android Devices Usage Behavior.

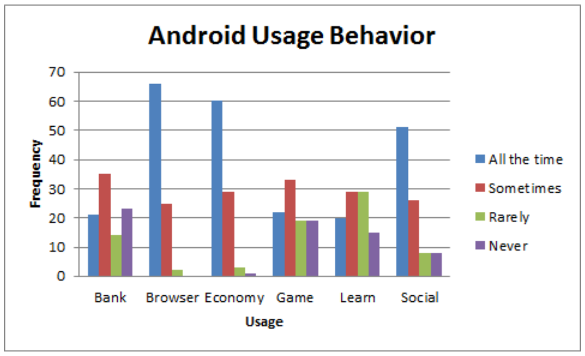


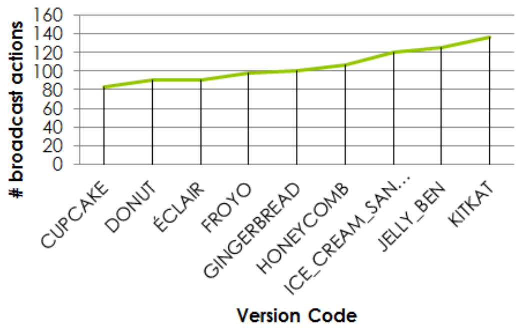
Table 44: The post survey results of 54 participants.

Factor	**Satisfaction (Mean)**
Overall Satisfaction	5.19
Time Satisfaction	5.37
Installation Satisfaction	5.76
App Navigation	5.74
Understanding the information	5.57
Learn about keyloggers	5.61
Surprising information	5.10
Apply Recomdendation	5.41
Enhance Security Awarness	5.37
Learn to prevent	5.55

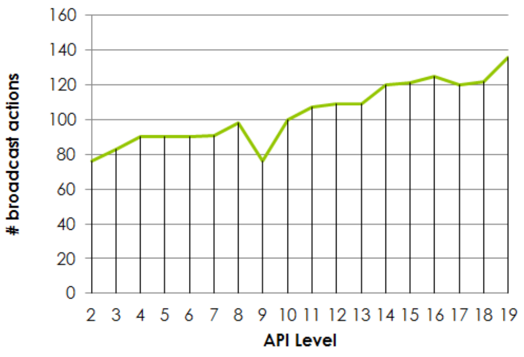
CHAPTER 7: BROADCAST RECEIVERS

A broadcast receiver is an Android component that enables applications to register for system or application events or actions(e.g. receive call, receive message). Once an event occurs, Android runtime notifies all registered receivers of that particular action. A broadcast is a message that any application can receive. The Android system delivers numerous broadcasts for system events, such as when an Internet connection is enabled or a new SMS arrives. While broadcast receivers are considered a useful feature by developers, users' experience and privacy can be affected negatively by them. Therefore, investigating Android broadcast receivers is vital. In this chapter, to understand the influence of broadcast receivers, we conduct an extensive study on the evolution of Android broadcast actions over all releases. We study the implications of including broadcast receivers in third party applications on users' privacy. We implement a malicious Android application in an effort to compromise users' privacy. The application uses one broadcast receiver which requires NORMAL permissions. The application collects some information and stores them in the cloud for future analysis. We perform two types of analysis: internal knowledge analysis and external knowledge analysis. In the internal knowledge analysis, we only used the collected data to infer some private information.The external knowledge analysis entails employing external data to gain more precise information about the victim.The study finds that the number of broadcasts actions have increased by 64%

since Android's first release.We also find that user privacy can be compromised using broadcast receivers that require NORMAL permissions.



(a) # of broadcast actions by Android version code.



(b) # of broadcast actions by Android API level.

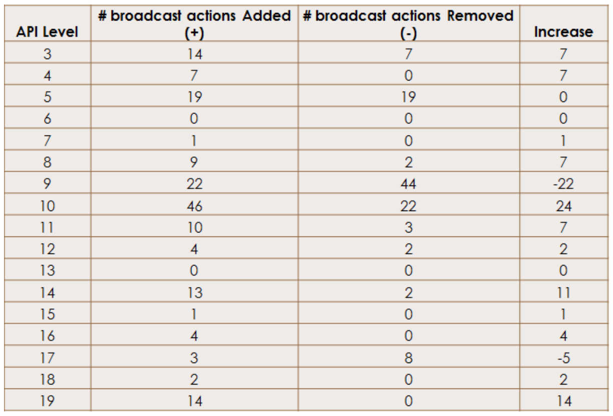
Figure 55: The evolution of Android broadcast actions over the 19 releases.

7.1 Android Broadcast Actions

We studied the Android platform to see how the set of broadcast actions has evolved. Figure 55(a) and Figure 55(b) show the increase in the number of actions over the version codes and API levels, respectively. In Figure 56 we show the number of broadcast actions that were added and removed on each release. While adding certain new actions can be justified as the need to fulfill new or hardware features (e.g. Bluetooth, Sensors), we couldn't find any explanation to removing some of these actions. In conclusion, the overall increase in the broadcast actions has given developers more control, but on the other hand, protecting users' privacy has become more challenging.

7.1.1 The Danger with Broadcast Receivers

In this section, we show why broadcast receivers can compromise users' privacy. We start by defining spy software, user assumptions and habits, then we demonstrate



API Level	# broadcast actions Added (+)	# broadcast actions Removed (-)	Increase
3	14	7	7
4	7	0	7
5	19	19	0
6	0	0	0
7	1	0	1
8	9	2	7
9	22	44	-22
10	46	22	24
11	10	3	7
12	4	2	2
13	0	0	0
14	13	2	11
15	1	0	1
16	4	0	4
17	3	8	-5
18	2	0	2
19	14	0	14

Figure 56: The change in the number of Android broadcast actions over the 19 APIs.

possible attack using broadcast receivers.

7.1.1.1 Android Spy Software

Android spy software is any software that aims at collecting as much data as possible from mobile devices. Some of this software is commercially distributed for sale. The companies that build such software present them as solutions to social problems, such as, monitoring children or employee activity. Attackers, on the other hand, use spy software to collect vital information, such as bank account info to gain financial leverage, or other data to blackmail the victim. The spy software enables the attacker to SILENTLY learn the truth about the victims' calls, text messages, GPS locations, and more. The silent monitoring is possible on Android with the help of broadcast receivers. An alternative solution would be to run services that would collect the data every defined period of time (e.g. 30 seconds). But this solution suffers from a major concern: a long-running service, once started, usually receives lower positioning in the list of background tasks over time, which would make it highly susceptible to be killed by the system.

7.1.2 User Assumptions and Habits

Over the years, users have developed certain habits and assumptions when dealing with malicious software. For instance, users know that they should not click on a link or attachment in an email that they don't trust, nor run untrusted executables, because doing so may infect their systems, which in turn compromises their privacy. An assumption of mobile devices users specifically is that installed applications can't access data if they are not on the foreground. Moreover, permissions give users an incomplete view of the capabilities each installed application has. A more accurate view would require retrieving registered Broadcast receivers of each application.

7.1.3 Attack Model

The attack presented here can be part of a legitimate application downloaded by the victim from an official or unofficial Android market. The application contains at least one activity, one service, and one broadcast receiver, although the application may contain some other components. A broadcast receiver needs an activity to be activated and registered. Moreover, the application may need to display an activity for the victim to interact with to hide the suspicious behind the scene behavior. The application has to provide some features to attract victims in the first place, and to keep it on the mobile device as long as possible.

7.1.4 Use Case Scenario

We developed an Android app *SpyApp* that has one *Activity*, which is the minimum we need for the broadcast receiver to get registered. The *SpyApp* registers for network related system events. To register for the above events, Android mandates two *NORMAL* permissions: *ACCESS_NETWORK_STATE* and *ACCESS_WIFI_STATE*. Whenever any of these events is fired, *SpyApp* would get notified by the system. At this moment, information about the network are getting collected, such as: *Status, MacAddress, SSID, STATE, TIME*. The information is then sent to a remote server by the app for future analysis. We performed two types of analysis on the collected data: internal knowledge analysis and external knowledge analysis. In the internal knowledge analysis, we only used the collected data to infer some knowledge about the victim. The external knowledge analysis entails employing external data to gain more defined knowledge about the victim, such as mapping SSIDs to a geographical location. Some of the private information include the victim's work place, their home, times she arrives and departs work, times she arrives and departs home, and other places she visits, like coffee shops.

7.2 Related Work

The work in [11] examined the inter-application communication in Android and presented several classes of potential attacks on applications, such as the risk of Broadcast theft and Broadcast injection. The dissertation proposed security tips to developers when implementing Broadcast receivers. Broadcast receivers can also be vulnerable to several attacks such as active denial-of-service attacks [46], which happen due to developers confusing inter-application and intra-application communication mechanisms. Additional related studies have shown that Android users pay poor attention to the permissions requests upon installing new apps [21], and thus alternate approaches were presented to provide users with the knowledge needed to make informed decisions about the applications they install [66].

7.3 Datasets Analysis

In this section we shed lights on the usage pattern of the Android Broadcast Receiver component by benign and malicious applications. We do so by analyzing the configurations files of large number of applications found in the literature.

7.3.1 The Description of the Datasets

The first dataset contains 1260 Android malware samples represent 49 different malware families [79]. The samples are manually and automatically collected from numerous Android Markets by the researchers at North Carolina State University over a course of fourteen months from August 2010 to October 2011. The second dataset contains 51179 benign Android applications and 4554 Android malware samples [40]. The malware samples were collected from malware repository websites such as VirusShare [], Contagio Mobile [], and Malware.lu [] during January to August 2013. The other Android applications were collected in the same period by downloading them from the Android market, Google Play, under the assumptions that they are benign.

7.3.2 The Analysis Focus

Both datasets have been analyzed for the usage of the statically registered Android Broadcast Receiver component that can be found in the configuration file called `AndroidManifest.xml`. We used a tool for reverse engineering Android `apk` files called `Apktool` to extract the `AndroidManifest.xml` files. We then parsed these files for Receivers components. We then summarized the results in different tables.

7.3.3 The Analysis Results of The Benign Dataset

The reverse engineering process of the 51,179 Android applications results in extracting only 48,262 manifest files. We found that 14,905 of these apps contain a receiver component in their manifest files and 33,341 does not contain a receiver component. Only 6020 of the 14,905 apps that contain one or more receiver components have a receiver to listen to system actions. Moreover, the 6020 apps contain 12,822 receivers for 134 distinct system actions. Table 45 shows the top most received system actions by the benign applications. Table 46 shows the moderately received system actions by the benign applications. Table 47 shows the low received system actions by the benign applications.

7.3.4 The Analysis Results of The Malware Dataset

The reverse engineering process of the 4554 and 1260 Android malware samples results in extracting 4542 and 1181 manifest files respectively. From a total of 5723 malware samples we found that 3930 of these apps contain a receiver component in their manifest files and 1793 does not contain a receiver component. We found that 3686 of the 3930 apps that contain one or more receiver components have a receiver to listen to a system action. Moreover, the 3686 apps contain receivers for 57 distinct system actions. Table 49 shows the top most received system actions by the benign applications. Table 50 shows the moderately received system actions by the benign applications. Table 51 shows the low received system actions by the benign applications.

Table 45: The number of receivers found for the system actions in the benign dataset (Top most received).

System Action	**Freq**
android.intent.action.BOOT_COMPLETED	4555
android.net.conn.CONNECTIVITY_CHANGE	834
android.intent.action.PACKAGE_REPLACED	602
android.intent.action.PACKAGE_ADDED	549
android.intent.action.PHON_STATE	484
android.provider.Telephony.SMS_RECEIVED	460
android.intent.action.PACKAGE_REMOVED	429
android.intent.action.TIME_SET	385
android.intent.action.TIMEZONE_CHANGED	345
android.intent.action.USER_PRESENT	326
android.intent.action.ACTION_SHUTDOWN	233
android.intent.action.NEW_OUTGOING_CALL	222
android.intent.action.MEDIA_BUTTON	197
android.net.wifi.WIFI_STATE_CHANGED	180
android.intent.action.DATE_CHANGED	173
android.intent.action.DEVICE_STORAGE_LOW	166
android.intent.action.ACTION_POWER_CONNECTED	162
android.intent.action.MEDIA_MOUNTED	151
android.intent.action.ACTION_POWER_DISCONNECTED	141
android.intent.action.LOCALE_CHANGED	141
android.intent.action.TIME_TICK	134
android.media.RINGER_MODE_CHANGED	123
android.bluetooth.adapter.action.STATE_CHANGED	117
android.intent.action.SCREEN_ON	114
android.net.wifi.STATE_CHANGE	102

Table 46: The number of receivers found for the system actions in the benign dataset (Moderate).

System Action	**Freq**
android.media.AUDIO_BECOMING_NOISY	91
android.intent.action.AIRPLANE_MODE	85
android.app.action.DEVICE_ADMIN_ENABLED	80
android.intent.action.PACKAGE_CHANGED	79
android.intent.action.SCREEN_OFF	78
android.intent.action.BATTERY_CHANGED	77
android.intent.action.PROVIDER_CHANGED	66
android.intent.action.MEDIA_UNMOUNTED	64
android.intent.action.PACKAGE_INSTALL	59
android.intent.action.CONFIGURATION_CHANGED	56
android.provider.Telephony.WAP_PUSH_RECEIVED	47
android.net.conn.BACKGROUND_DATA_SETTING_CHANGED	46
android.intent.action.HEADSET_PLUG	45
android.intent.action.EXTERNAL_APPLICATIONS_AVAILABLE	45
android.bluetooth.device.action.ACL_CONNECTED	44
android.bluetooth.device.action.ACL_DISCONNECTED	42
android.intent.action.PACKAGE_RESTARTED	41
android.net.wifi.supplicant.CONNECTION_CHANGE	39
android.intent.action.MEDIA_EJECT	39
android.intent.action.CAMERA_BUTTON	37
android.intent.action.MEDIA_SCANNER_FINISHED	36
android.intent.action.MEDIA_REMOVED	31
android.net.wifi.SCAN_RESULTS	28
android.intent.action.BATTERY_LOW	27
android.intent.action.DOCK_EVENT	27
android.net.wifi.supplicant.STATE_CHANGE	26
android.intent.action.UMS_CONNECTED	23
android.bluetooth.intent.action.BLUETOOTH_STATE_CHANGED	22
android.intent.action.BATTERY_OKAY	21
android.intent.action.UMS_DISCONNECTED	19
android.intent.action.DOWNLOAD_COMPLETE	19
android.intent.action.DEVICE_STORAGE_OK	18
android.intent.action.MY_PACKAGE_REPLACED	18
android.intent.action.DATA_SMS_RECEIVED	16
android.intent.action.PACKAGE_DATA_CLEARED	15
android.intent.action.WALLPAPER_CHANGED	15
android.intent.action.MEDIA_BAD_REMOVAL	15
android.intent.action.MEDIA_SHARED	15
LikeAbove(3).EXTERNAL_APPLICATIONS_UNAVAILABLE	14
android.intent.action.REBOOT	13
android.intent.action.PACKAGE_FULLY_REMOVED	12

Table 47: The number of receivers of the system actions found in the benign dataset (Low).

System Action	**Freq**
android.intent.action.DOWNLOAD_NOTIFICATION_CLICKED	9
android.intent.action.PACKAGE_FIRST_LAUNCH	8
android.intent.action.MEDIA_CHECKING	8
android.net.wifi.RSSI_CHANGED	8
android.intent.action.MEDIA_SCANNER_STARTED	7
android.intent.action.GTALK_CONNECTED	7
android.bluetooth.headset.action.STATE_CHANGED	7
android.media.VIBRATE_SETTING_CHANGED	7
android.bluetooth.intent.action.REMOTE_DEVICE_CONNECTED	7
android.bluetooth.intent.action.SCAN_MODE_CHANGED	7
android.bluetooth.intent.action.REMOTE_NAME_UPDATED	6
android.intent.action.MEDIA_UNMOUNTABLE	6
android.media.SCO_AUDIO_STATE_CHANGED	6
android.intent.action.MEDIA_NOFS	6
android.bluetooth.intent.action.REMOTE_DEVICE_DISCONNECTED	6
LikeAbove(4).REMOTE_DEVICE_DISCONNECT_REQUESTED	5
android.bluetooth.device.action.FOUND	5
android.bluetooth.a2dp.action.SINK_STATE_CHANGED	5
android.intent.action.INPUT_METHOD_CHANGED	5
android.bluetooth.intent.action.NAME_CHANGED	5
android.bluetooth.intent.action.PAIRING_REQUEST	5
android.bluetooth.intent.action.PAIRING_CANCEL	5
android.bluetooth.intent.action.REMOTE_DEVICE_FOUND	5
android.bluetooth.intent.action.DISCOVERY_STARTED	5
android.bluetooth.intent.action.DISCOVERY_COMPLETED	5

Table 48: The number of receivers of the system actions found in the benign dataset (Very Low).

System Action	**Freq**
android.bluetooth.device.action.ACL_DISCONNECT_REQUESTED	4
android.bluetooth.adapter.action.SCAN_MODE_CHANGED	4
android.bluetooth.device.action.BOND_STATE_CHANGED	4
LikeAbove(2).headset.profile.action.CONNECTION_STATE_CHANGED	4
android.net.wifi.NETWORK_IDS_CHANGED	4
android.bluetooth.adapter.action.DISCOVERY_STARTED	3
android.bluetooth.intent.action.HEADSET_STATE_CHANGED	3
android.intent.action.CONTENT_CHANGED	3
android.bluetooth.adapter.action.DISCOVERY_FINISHED	3
android.bluetooth.a2dp.intent.action.SINK_STATE_CHANGED	3
LikeBelow(4).VENDOR_SPECIFIC_HEADSET_EVENT	2
android.bluetooth.headset.action.AUDIO_STATE_CHANGED	2
android.speech.tts.TTS_QUEUE_PROCESSING_COMPLETED	2
android.intent.action.MEDIA_SCANNER_SCAN_FILE	2
android.bluetooth.a2dp.profile.action.CONNECTION_STATE_CHANGED	2
android.media.action.OPEN_AUDIO_EFFECT_CONTROL_SESSION	2
com.google.gservices.intent.action.GSERVICES_CHANGED	2
android.net.wifi.p2p.CONNECTION_STATE_CHANGE	2
android.intent.action.PACKAGE_NEEDS_VERIFICATION	2
android.bluetooth.intent.action.REMOTE_DEVICE_DISAPPEARED	2
android.intent.action.GTALK_DISCONNECTED	2
android.intent.action.DREAMING_STOPPED	1
android.provider.Telephony.SIM_FULL	1
android.media.action.CLOSE_AUDIO_EFFECT_CONTROL_SESSION	1
android.bluetooth.intent.action.REMOTE_NAME_FAILED	1
android.bluetooth.a2dp.profile.action.PLAYING_STATE_CHANGED	1
android.intent.action.UID_REMOVED	1
android.net.wifi.p2p.STATE_CHANGED	1
android.bluetooth.headset.profile.action.AUDIO_STATE_CHANGED	1
android.bluetooth.adapter.action.LOCAL_NAME_CHANGED	1
android.bluetooth.intent.action.BOND_STATE_CHANGED_ACTION	1
android.intent.action.MANAGE_PACKAGE_STORAGE	1
android.app.action.ACTION_PASSWORD_FAILED	1
android.bluetooth.intent.action.DISABLED	1
android.intent.action.NEW_VOICEMAIL	1
android.media.ACTION_SCO_AUDIO_STATE_UPDATED	1
android.hardware.action.NEW_VIDEO	1
android.intent.action.PROXY_CHANGE	1
android.bluetooth.device.action.NAME_CHANGED	1
android.bluetooth.intent.action.ENABLED	1
android.app.action.ACTION_PASSWORD_SUCCEEDED	1
android.bluetooth.intent.action.HEADSET_ADUIO_STATE_CHANGED	1

Table 49: The number of receivers found for the system actions in the malware dataset (Top most received).

System Action	**Freq**
android.intent.action.BOOT_COMPLETED	3468
android.intent.action.DATA_SMS_RECEIVED	967
android.provider.Telephony.SMS_RECEIVED	795
android.intent.action.PHONE_STATE	514
android.intent.action.USER_PRESENT	513
android.net.conn.CONNECTIVITY_CHANGE	435
android.intent.action.ACTION_POWER_CONNECTED	278
android.intent.action.INPUT_METHOD_CHANGED	276
android.intent.action.PACKAGE_ADDED	192
android.intent.action.UMS_CONNECTED	188
android.intent.action.UMS_DISCONNECTED	187

Table 50: The number of receivers found for the system actions in the malware dataset (moderate).

System Action	**Freq**
android.provider.Telephony.WAP_PUSH_RECEIVED	86
android.intent.action.NEW_OUTGOING_CALL	74
android.intent.action.PACKAGE_REMOVED	68
android.net.wifi.WIFI_STATE_CHANGED	62
android.intent.action.BATTERY_LOW	46
android.intent.action.BATTERY_OKAY	46
android.provider.Telephony.SIM_FULL	45
android.intent.action.PACKAGE_REPLACED	36
android.intent.action.PACKAGE_CHANGED	23
android.intent.action.TIME_SET	19
android.intent.action.PACKAGE_INSTALL	17
android.intent.action.PACKAGE_RESTARTED	16

Table 51: The number of receivers found for the system actions in the malware dataset (low).

System Action	**Freq**
android.intent.action.MEDIA_MOUNTED	8
android.intent.action.MEDIA_REMOVED	7
android.intent.action.MEDIA_EJECT	6
android.intent.action.MEDIA_UNMOUNTED	6
android.intent.action.MEDIA_NOFS	5
android.intent.action.MEDIA_BAD_REMOVAL	5
android.app.action.DEVICE_ADMIN_ENABLED	4
android.intent.action.BATTERY_CHANGED	3
android.intent.action.ACTION_SHUTDOWN	3
android.bluetooth.intent.action.BLUETOOTH_STATE_CHANGED	2
android.intent.action.DOCK_EVENT	2
android.bluetooth.intent.action.SCAN_MODE_CHANGED	2
android.intent.action.DATE_CHANGED	2
android.intent.action.SCREEN_OFF	1
android.intent.action.CAMERA_BUTTON	1
android.bluetooth.intent.action.PAIRING_REQUEST	1
android.bluetooth.intent.action.REMOTE_NAME_UPDATED	1
LikeAbove(4).REMOTE_DEVICE_DISCONNECT_REQUESTED	1
android.bluetooth.adapter.action.SCAN_MODE_CHANGED	1
android.intent.action.GTALK_CONNECTED	1
android.bluetooth.intent.action.REMOTE_DEVICE_DISCONNECTED	1
android.intent.action.SCREEN_ON	1
android.bluetooth.adapter.action.STATE_CHANGED	1
android.intent.action.ACTION_POWER_DISCONNECTED	1
android.bluetooth.intent.action.NAME_CHANGED	1
android.intent.action.TIME_TICK	1
android.bluetooth.intent.action.REMOTE_DEVICE_CONNECTED	1
android.net.conn.BACKGROUND_DATA_SETTING_CHANGED	1
android.bluetooth.intent.action.DISCOVERY_COMPLETED	1
android.bluetooth.intent.action.DISCOVERY_STARTED	1
android.bluetooth.intent.action.PAIRING_CANCEL	1
android.bluetooth.intent.action.REMOTE_DEVICE_FOUND	1

Table 52: A summary of the dataset that we analyzed for the Broadcast receivers information.

Dataset	Size	Has Receivers (%)	No Receivers (%)	Sys Receivers (%)
Benign	48262	14905 (31)	33341 (69)	6020 (12)
Malicious	5723	3930 (69)	1793 (31)	3686 (64)

7.3.5 Summary

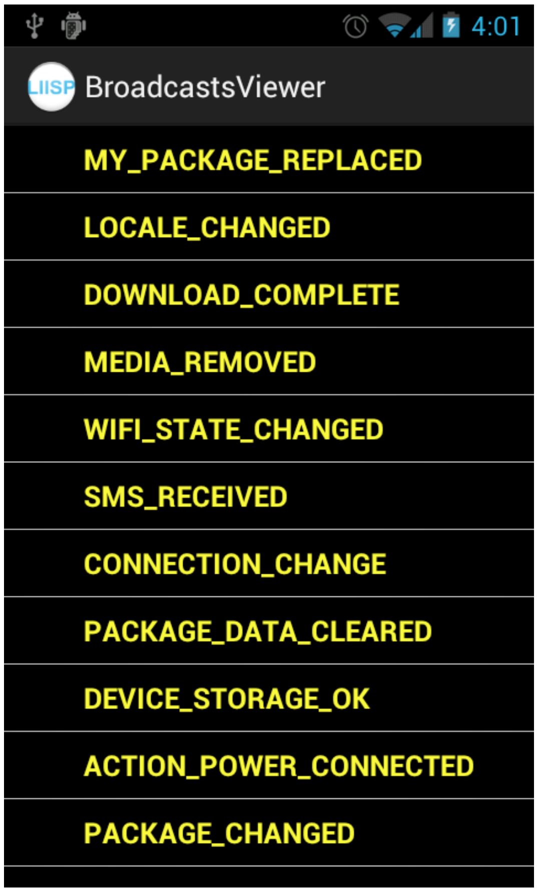
Table 52 shows a summary of the difference in receivers usage between benign and malware applications. Only 31% of the benign applications contain a receiver component and only 12% contain a receiver that listen to a system broadcast action. Whereas the analysis shows that 69% of the malware samples contain a receiver component and 64% contain a receiver that listen to system broadcast action. The huge difference in usage between the two brings us to a conclusion that system broadcast actions found to be useful to malwares; thus, it must be carefully granted.

7.4 Proposed Solutions

Currently on Android, when user installs a new app, he/she would be prompted to grant it some permissions. The prompt doesn't include any information about the broadcast receivers. Moreover, the existing tools that assess the risk of installed apps focus on the permissions they possess. Thus, we propose to

- Develop a tool to display the registered broadcast receivers by installed applications.

- Propose to modify the Android package installer to change the permission prompts to include broadcast receivers.



BroadcastsViewer

MY_PACKAGE_REPLACED

LOCALE_CHANGED

DOWNLOAD_COMPLETE

MEDIA_REMOVED

WIFI_STATE_CHANGED

SMS_RECEIVED

CONNECTION_CHANGE

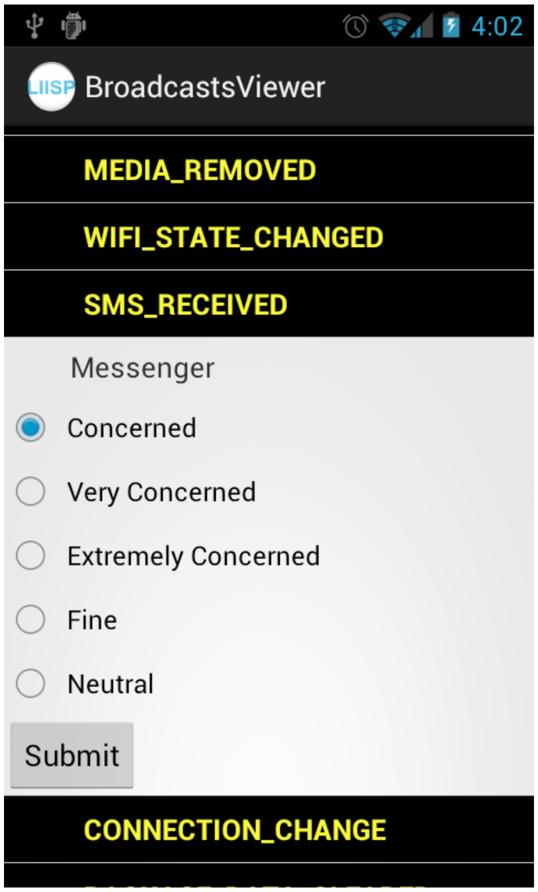
PACKAGE_DATA_CLEARED

DEVICE_STORAGE_OK

ACTION_POWER_CONNECTED

PACKAGE_CHANGED

(a) Main Page.



BroadcastsViewer

MEDIA_REMOVED

WIFI_STATE_CHANGED

SMS_RECEIVED

Messenger

[✓ Concerned]

Very Concerned

Extremely Concerned

Fine

Neutral

Submit

CONNECTION_CHANGE

(b) User Feedback Form.

Figure 57: BroadcastsViewer Tool for Detecting and Displaying Registered System Broadcast Actions.

7.4.1 Android Package Installer

PackageInstaller is the default Android application to interactively install new applications (apps). It provides the needed interfaces to allow the user to manage the installation process. PackageInstaller calls *InstallAppProgress* activity to receive instructions from the user. InstallAppProgress asks a *PackageManager* service to install the app via *installd*. The installd daemon's main role is to receive requests from the PackageManager service via the Linux domain socket. The installd executes a series of steps to install APKs with root permission. The source code of the

PackageInstaller can be found online [27]. It has many components, such as *PackageInstallerActivity.java, PackageUtil.java, InstallAppProgress.java.* We downloaded the Android source code for the Kitkat version of PackageInstaller. Currently, we are working on modifying the installer to retrieve and display the broadcast receivers information as part of the permissions prompt.

7.4.2 BroadcastsViewer Tool

Figure 57 shows *BroadcastsViewer*, the tool we developed to detect registered system broadcast actions by the installed apps. Our focus is on the system Broadcasts actions which would make our contribution unique compared to previous works. *BroadcastsViewer* will also provide the user with some statistics on the usage of each broadcast receiver. The user can also give his/her feedback on their level of concern for each receiver action registered to each app. Aggregating apps' receiver usage can help us draw conclusions on what is suspicious and what is not. For example, an app like Dropbox registers to receive the CONNECTIVITY_CHANGE action, and it would be justifiable given the cloud storage service it provides and its dependency on connectivity to accomplish this.

CHAPTER 8: CONCLUSION

8.1 Dissertation Contributions

The different OAuth implementations approaches in mobile applications adopted by popular resource providers are identified and possible attacks are being demonstrated by analyzing the source of 18 SDKs once in 2013 and another time in 2015. We summarize the OAuth implementation choices made by the service providers in their SDKs (Software Development Kits) and by developers in their OAuth-Embedded mobile applications by analyzing the source codes of 430 applications collected from the Google Play Store. In-lab user study of 29 participants and online user study of 430 participants are conducted to evaluate users' awareness and comprehension to the OAuth-Embedded mobile applications and the possible attacks. Seven new security cue designs for WebView-based mobile applications are proposed, 4 for the authentication page and 3 for the authorization page, in addition to design found in the literature, and evaluated on observability, understandability and affectivity aspects. After realizing the problem of OAuth implementations in mobile applications a stand-alone application-based solution called OAuthManager is proposed and a prototype is implemented. The solution is based on the concept of privilege separation and does not require high overhead. However, the solution mandates that both developers and service providers change their implementations to work with the proposed solution. Thus, we introduce *SecureOAuth*, a whitelist access control protection framework for the Android platform. SecureOAuth is composed of: Android library modifications, service creation, and system app creation. A prototype of the SecureOAuth framework is implemented and evaluated on performance and memory overhead. The framework hardens the OAuth-WebView implementation with bounded overhead while keeping the user's involvement to minimum. Moreover, the framework requires no implementations' changes and assumes strong attacking assumptions. An analysis study of a set of 100 third-party Android keyboards, collected from the market, is conducted. The number and type of permissions that are requested by these keyboards make them potential victims for the keylogger attack. The users and keyboard developers roles in increasing/decreasing the chance of successful keylogger attack is also studied by conducting an online study of 92 participants and a questionnaire for 10 developers. The study shows that keylogging threat is of high probability due to the current security configurations and users and developers choices. Moreover, the study shoes that the risk can be reduced by educating the users and by adopting new development approaches.As far as the broadcast receivers, the evaluation of Android broadcast actions is studied and an attack scenario that made possible by the broadcast receivers is demonstrated. Moreover, an analysis study is conducted on a set of 48262 benign applications and 5723 malware samples. The analysis results showed 69% of the malicious applications contain a a receiver component, 64% contain a receiver for a system action broadcast. Of the 48262 benign application, only 31% of them contain a receiver component, and only 12% of them contain a receiver component for a system action broadcast.

8.2 Limitations and Future Work

Like the majority of researches that have been conducted on Android security we mainly used free apps to conduct our analysis. Including paid apps in the datasets would make the results more robust and accurate. Our work on SecureOAuth did not offer any mechanism for keeping and maintaining popular service providers' OAuth URLs in a remote server that is accessible by the SecureOAuth app and the service providers. Populating the ProtectedURLsDB at the first run can be done manually by the manufacturer but the subsequent changes and modifications must be done automatically; thus, there is a need to have this mechanism in hand. Moreover, there are cases where a malicious app might claim to redirect the user to an OAuth login page, but instead redirects the user to a local file or remote server that the attacker controls, similar to a phishing attack. Since the WebView has no URL bar, it is hard for the users to know if they are actually on an OAuth page or not. The current SecureOAuth implementation does not consider such an attack, we plan to extend our work to cover these cases. Besides that, there are numerous future studies that can be conducted to extend it. An extension to this work would be to conduct a user study on the Android Broadcast receivers to understand the user and developers perspectives. The user perspective would shed lights on her awareness and comprehension to the receivers as a process running in the background and continuously listening to the system's broadcasts. The developer's perspective would give us insights on the need to have statically registered Broadcast receivers instead of dynamic ones. Moreover, in this dissertation, we proposed and tested the efficiency of a number of security cue designs to alert users of OAuth related violations. Another future work would be to propose new permission prompt designs that incorporates the Broadcast receivers information, then to study its effect on users decision to install apps on their mobile devices. As far as the keylogging threat, a future work could be to look at more sophisticated attacking scenarios in which two or more apps are colluding to keylogs users' inputs. One app could be responsible for stealing users' typing and storing them in the device, the mission of the other app would be to send these data into a remote server. In this, dissertation, we identified many ways by which two apps running on the same device can exchange data. Some of these ways can be discovered by scanning their configurations files whereas others require more complicated means.

REFERENCES

[1] ANDROID OS. "http:\\www.android.com", 2016.

[2] APPLE iOS9. "http:\\www.apple.com/ios", 2016.

[3] A. Wulf. Stealing Passwords is Easy in Native Mobile Apps Despite OAuth. http:\\welcome.totheinter\\.net/2011/01/12/stealing-passwords-is-easy-in-native-mobile-apps-\\despite-oauth/. Accessed: 09/12/2014.

[4] Android. Cupcake. "http:\\developer.android.com/about/versions\android-1.5-highlights.html/", 2013.

[5] Android. Investigating Your RAM Usage. "https:\\developer.android.com/tools/debugging/debugging-memory.html", September 2014.

[6] Android. ART and Dalvik. https:\\source.android.com/devices/tech/dalvik/, 2015.

[7] AppBrain. Appbrain statistics. "http:\\www.appbrain.com/", 2013.

[8] Box. Box SDK for Android (version 2.0). https:\\github.com/box/box-android-sdk, 2013.

[9] Box. Box SDK for iOS (version 1.0). https:\\github.com/box/box-ios-sdk, 2013.

[10] E. Chen, Y. Tian, Y. Pei, R. Kotcher, S. Chen, and P. Tague. Oauth demystified for mobile application developers. In *ACM Conference on Computer and Communications Security (CCS)*. to appear.

[11] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[12] E. Chin and D. Wagner. Bifocals: Analyzing webview vulnerabilities in android applications. In *Information Security Applications - 14th International Workshop, WISA 2013, Jeju Island, Korea, August 19-21, 2013, Revised Selected Papers*, pages 138–159, 2013.

[13] J. Cho, G. Cho, and H. Kim. Keyboard or keylogger?: a security analysis of third-party keyboards on android. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, PST '15, Izmir, Turkey, 2015. ACM.

[14] D. Recordon and B. Fitzpatrick. OpenID Authentication 2.0. http:\\openid.net/specs/openid-authentication-2_0.html, 2007.

[15] Inserting keylogger code in Android SwiftKey using apktool, 2013. URL: `http:\\www.somewebpage.org/` [accessed: 2015-03-09].

[16] Q. Do, B. Martini, and K.-K. R. Choo. Exfiltrating data from android devices. *Computers & Security*, 48:74–91, 2015.

[17] Q. Do, B. Martini, and K. R. Choo. Exfiltrating data from android devices. *Computers & Security*, 48:74–91, 2015.

[18] Dropbox. Core API Development kits and documentation (version 1.5.4). `https:\\www.drop$box.com/developers/core/sdk`, 2013.

[19] Facebook. Facebood SDK for android. `https:\\github.com/facebook/facebook-android-sdk`, 2012.

[20] Facebook. The Facebook SDK for iOS (version 3.5.1). `https:\\developers.facebook.com/ios/`, 2013.

[21] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, pages 3:1–3:14, New York, NY, USA, 2012. ACM.

[22] A. P. Felt, H. J. Wang, and A. Moshchuk. Permission re-delegation: Attacks and defenses. In *Proceedings of the 20th USENIX conference on Security*, 2011.

[23] A. A. Ghorbani, V. Torra, H. Hisil, A. Miri, A. Koltuksuz, J. Zhang, M. Sensoy, J. García-Alfaro, and I. Zincir, editors. *13th Annual Conference on Privacy, Security and Trust, PST 2015, Izmir, Turkey, July 21-23, 2015*. IEEE, 2015.

[24] Github. APK-Multi-Tool. `https:\\github.com/APK-Multi-Tool/APK-Multi-Tool`, 2015.

[25] N. Goldshlager. How i hacked any facebookaccount...again!. `"http:\\www.breaksec.com/?p=5753"`, March 2013.

[26] N. Goldshlager. How I Hacked Facebook OAuth To Get Full Permission On Any Facebook Account (Without App "Allow" Interaction) - Break Security. `"http:\\homakov.blogspot.ca/2013/02/hacking-facebook-with-oauth2-and-chrome.html"`, March 2013.

[27] Google. PackageInstaller Source Code. `packages/apps/PackageInstaller/src/com/android/packageinstaller`.

[28] Google. Google's AuthSub authentication. `http:\\code.google.com/apis/accounts/docs/AuthSub.html`, 2008.

[29] Google. Android categoration. `"https:\\support.google.com/googleplay/android-developer/answer/113475?hl=en/"`, 2013.

[30] Google. Google+ Platform for Android (version 1.3.0). `https:\\developers.google.com/+/mobile/android/getting-started`, 2013.

[31] Google. Google+ Platform for iOS (version 1.3.0). `https:\\developers.google.com/+/mobile/ios/`, 2013.

[32] Google. Remote debugging on Android with Chrome DevTools. `https:\\developers.google.com/web/tools/chrome-devtools/debug/remote-debugging/remote-debugging?hl=en`, 2015.

[33] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: Scalable and accurate zero-day android malware detection. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 281–294, New York, NY, USA, 2012. ACM.

[34] B. Gruver. Smali/baksmali. `https:\\github.com/JesusFreke/smali`, 2015.

[35] E. Hammer-Lahav. Oauth security advisory. `"http:\\oauth.net/advisories/2009-1/"`, April 2009.

[36] E. Homakov. How we hacked facebook with oauth2 and chrome bugs. `"http:\\homakov.blogspot.ca/2013/02/hacking-facebook-with-oauth2-and-chrome.html"`, February 2013.

[37] E. Homakov. Oauth1, oauth2, oauth...?. `"http:\\homakov.blogspot.ca/2013/03/oauth1-oauth2-oauth.html"`, March 2013.

[38] Creating an Input Method, 2013. URL: `http:\\developer.android.com/guide/topics/text\creating-input-method.html/` [accessed: 2015-03-09].

[39] SoftKeyboard Sample, 2013. URL: `https:\\android.googlesource.com/platform/development/+/master/samples\SoftKeyboard/` [accessed: 2015-03-09].

[40] A. M. Hyunjae Kang, Jae-wook Jang and H. K. Kim. Detecting and Classifying Android Malware Using Static Analysis along with Creator Information,. *International Journal of Distributed Sensor Networks*, 2015(479174):9, 2015.

[41] Instagram. Instagram iOS Authentication (version 1). `http:\\instagram.com/developer/authentication/#/`, 2013.

[42] Instgram. Instagram client for Android (version 1.86). `https:\\github.com/markchang/android-instagram/`, 2013.

[43] Internet Engineering Task Force (IETF). The OAuth 1.0 Authorization Framework. `"https:\\tools.ietf.org/html/rfc5849"`, October 2010.

[44] Internet Engineering Task Force (IETF). The OAuth 2.0 Authorization Framework. `"http:\\www.rfc-editor.org/rfc/rfc6749.txt"`, October 2012.

[45] Java. Java technology. `https:\\www.ja$va.com/en/about/`, 2015.

[46] D. Kantola, E. Chin, W. He, and D. Wagner. Reducing attack surfaces for intra-application communication in android. Technical Report UCB/EECS-2012-182, EECS Department, University of California, Berkeley, Jul 2012.

[47] Linkedin. A java wrapper for linkedin API (version 1.0.429). `http:\\code.google.com/p/linkedin-j/`, 2013.

[48] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin. Attacks on webview in the android system. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, New York, NY, USA, 2011. ACM.

[49] T. Luo, X. Jin, A. Ananthanarayanan, and W. Du. Touchjacking attacks on web in android, ios, and windows phone. In *Foundations and Practice of Security - 5th International Symposium, FPS 2012, Montreal, QC, Canada, October 25-26, 2012, Revised Selected Papers*, pages 227–243, 2012.

[50] M. McGloin and P. Hunt. OAuth 2.0 Threat Model and Security Considerations. `http:\\tools.ietf.org/id/draft-ietf-oauth-v2-threatmodel-00.txt/`, 2011.

[51] Microsoft. Microsoft Live Connect. `http:\\msdn.microsoft.com/en-us/windowslive/default.aspx`, 2010.

[52] Microsoft. The Live SDK for Android library (version 5.0). `https:\\github.com/liveservices/LiveSDK-for-Android`, 2013.

[53] Microsoft. The Live SDK for iOS library (version 5.0). `https:\\github.com/liveservices/LiveSDK-for-ios`, 2013.

[54] F. Mohsen and M. Shehab. Android Keylogging Threat. In *Proceedings of the 9th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing (COLLABORATECOM) Oct 20-23, 2013, Austin, Texas*, pages 545–552, 2013.

[55] A. Mylonas and A. Kastania. Delegate the smartphone user? Security awareness in smartphone platforms. `http:\\www.science$direct.com/science/article/pii/S0167404812001733`, 2013.

[56] OAuth 2.0. The OAuth 2.0 Protocol. `"http:\\tools.ietf.org/html/draft-ietf-oauth-v2-10"`, 2010.

[57] Open Source. A java flickr API library (version 2.0). `https:\\github.com/lukhnos/objectiveflickr/`, 2013.

[58] Open Source. A java flickr API library (version 2.0.0). `http:\\code.google.com/p/flickrj-android/wiki/HowToGuide4Android/`, 2013.

[59] Open Source. API Kits (version 1.1). http:\\www.whitn$eyland.com/2011/03/iphone-oauth.html, 2013.

[60] Open Source. Unofficial Java library for the Twitter API (version 3.0.4-SNAPSHOT). https:\\github.com/yusuke/twitter4j/, 2013.

[61] Oracle. keytool - Key and Certificate Management Tool. https:\\docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html, 2015.

[62] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner. AdDroid: Privilege separation for applications and advertisers in Android. In *Proceedings of AsiaCCS*, May 2012.

[63] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, SSYM'03, Berkeley, CA, USA, 2003. USENIX Association.

[64] R. Paul. Compromising Twitter's OAuth security system. http:\\arstechnica.com/security/2010/09/twitter-a-case-study-on-how-to-do-oauth-wrong/.

[65] R. Wang, Y. Zhou,S. Chen, S. Qadeer, D. Evans, Y. Gurevich. Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization . http:\\research.microsoft.com/pubs/188979/ExplicatingSDKs-TR.pdf/, 2013.

[66] S. Rosen, Z. Qian, and Z. M. Mao. Appprofiler: A flexible method of exposing privacy-related behavior in android applications to end users. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, pages 221–232, New York, NY, USA, 2013. ACM.

[67] S. Pai and Y. Sharma and S. Kumar and R. Pai and S. Singh. Formal Verification of OAuth 2.0 Using Alloy Framework. pages 655–659, June 2011.

[68] S. Shekhar, M. Dietz, and D. S. Wallach. Adsplit: separating smartphone advertising from applications. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, Berkeley, CA, USA, 2012.

[69] S.-T. Sun and K. Beznosov. The devil is in the (implementation) details: An empirical analysis of oauth sso systems. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 378–390, New York, NY, USA, 2012. ACM.

[70] C. Tumbleson. Apktool. https:\\ibotpeaches.github.io/Apktool/, 2015.

[71] Twitter-iOS. Twitter iOS Integration (version 1.1). https:\\dev.twitter.com/docs/ios/, 2013.

[72] W. Enck and P. Gilbert and B. Chun and L. Cox and J. Jung and P. McDeniel and A. Sheth . TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. pages 1–6, 2010.

[73] R. Wang, S. Chen, and X. Wang. Signing me onto your accounts through facebook and google: a traffic-guided security study of commercially deployed single-sign-on web services. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*. IEEE Computer Society, May 2012.

[74] R. Wang, L. Xing, X. Wang, and S. Chen. Unauthorized origin crossing on mobile platforms: Threats and mitigation. ACM Conference on Computer and Communications Security (ACM CCS), November 2013.

[75] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich. Explicating sdks: Uncovering assumptions underlying secure authentication and authorization. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 399–414, Berkeley, CA, USA, 2013. USENIX Association.

[76] Wiki. Android-Adding SystemService. `"http:\\processors.wiki.ti.com/index.php/Android-Adding_SystemService"`, September 2014.

[77] WIKIPEDIA. ios 8. `"http:\\en.wikipedia.org\wiki/IOS\_8"`, 2015.

[78] Yahoo Inc. Yahoo Browser-based Authentication. `http:\\developer.yahoo.com/auth`, 2008.

[79] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pages 95–109, 2012.

Appendix

A Applying the Solution Patch

To apply the patch:

1. Download a fresh copy of android source code from AOSP git. version (android 4.4.2_r1) as we have created a patch on top of it.

2. Go to the root folder of the code and paste the patch file inside it.(the patch file should be on the same level as the frameworks , packages and other folders in android source code).

3. Then, apply the patch by running the command.

   patch -p1 ¡ SecureOAuth.patch

4. build the source code.

You must not get any HUNK or reject statements on the command line while applying. If so, the patch did not apply properly.

To revert the patch:

1. patch -Rp1 ¡ SecureOAuth.patch

Please follow the above instructions carefully. Also please note that patches would create problems if not applied on the same version of android source code that was created. (We had created it on android kitkat version 4.4.2_r1)

B To Download the original source code

```
repo init —u https://android.googlesource—
.com/platform/manifest —b android—4.4.2/_r1
```

C The new/modified files and their paths

The modified files:

```
Build/target/product/generic_no_telephony.mk
External/chromium_org/content/browser/
android/content_view_core_impl.cc
External/chromium_org/content/browser/
android/content_view_core_impl.h
```

```
Frameworks/base/services/java/com/android/

server/SystemServer.java

Frameworks/base/Android.mk
```

The new files:

```
Packages/apps/SecureOAuth

Frameworks/base/core/java/android/os/

ISecureOAuthService.aidl
```

DISSERTATION PUBLICATIONS

[1] Fadi Mohsen and Mohamed Shehab (2013). Android Keylogging Threat. 9th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing (Collaborate).

[2] Mohamed Shehab and Fadi Mohsen (2014). Towards Enhancing the Security of OAuth Implementations In Smart Phones. IEEE International Conference on Mobile Services (MS).

[3] Fadi Mohsen, Mohamed Shehab, Emmanuel Bello-Ogunu, and Abeer Al Jarrah (2014). Android System Broadcast Actions Broadcasts Your Privacy. (Poster). 21st ACM Conference on Computer and Communications Security (CCS).

[4] Mohamed Shehab and Fadi Mohsen (2014). Secure OAuth Implementations in Smart Phones. (Poster). 4th ACM Conference on Data and Application Security and Privacy (CODASPY).

[5] Fadi Mohsen, Emmanuel Bello-Ogunu, and Mohamed Shehab (2016). Investigating the Keylogging threat in Android - User Perspective. 2nd IEEE International Conference on Mobile and Secure Services (MobiSecServ).

[6] Q.Duan, Y.Wang, Fadi Mohsen, and E.Al-Shaer (2013). Private and Anonymous Data Storage and Distribution in Cloud. IEEE 10th International Conference on Services Computing (SCC).

[7] Mohammad Rahman, Fadi Mohsen and Ehab Al-Shaer (2013). A Formal Model for Sustainable Vehicle-to-Grid Management. ACM Workshop on Smart Energy Grid Security (SEGS).