12-2015

All your sessions are belong to us: Investigating authenticator leakage through backup channels on Android

Guangdong BAI

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Jianliang WU

Quanqi YE

Li LI

See next page for additional authors

All Your Sessions are Belong to us: Investigating Authenticator Leakage through Backup Channels on Android

Guangdong Bai[1,2], Jun Sun[2], Jianliang Wu[3], Quanqi Ye[1], Li Li[1,2], Jin Song Dong[1], and Shanqing Guo[3]
[1]*National University of Singapore*
[2]*Singapore University of Technology and Design*
[3]*Shandong University*

***Abstract*—Security of authentication protocols heavily relies on the confidentiality of credentials (or *authenticators*) like passwords and session IDs. However, unlike browser-based web applications for which highly evolved browsers manage the authenticators, Android apps have to construct their own management. We find that most apps simply locate their authenticators into the persistent storage and entrust underlying Android OS for mediation. Consequently, these authenticators can be leaked through compromised backup channels. In this work, we conduct the first systematic investigation on this previously overlooked attack vector. We find that nearly all backup apps on Google Play inadvertently expose backup data to any app with internet and SD card permissions. With this exposure, the malicious apps can steal other apps' authenticators and obtain complete control over the authenticated sessions. We show that this can be stealthily and efficiently done by building a proof-of-concept app named AuthSniffer. We find that 80 (68.4%) out of the 117 tested top-ranked apps which have implemented authentication schemes are subject to this threat. Our study should raise the awareness of app developers and protocol analysts about this attack vector.**

I. Introduction

Due to the astonishing popularity of Android, web applications often have Android applications (or *app*s) as their companion clients. These *web-based apps* deliver services in a more accessible manner than their web browser based counterparts and therefore have attracted a large number of active visitors from mobile devices. Taking the social networking service as an example, users of Facebook mobile clients have surpassed its desktop users [18].

Typically, the web-based apps play the role of the traditional web browsers, which fetch resources from the web servers. The apps thus implement compatible authentication protocols to enable the users to log into their web accounts. A typical authentication process involves some security-critical credentials (denoted by *authenticators*), such as passwords, session IDs, secret cookies and OAuth tokens. These authenticators are the backbones of the authentication protocols and deserve a secure management. However, unlike the well-evolved web browsers which have employed a series of advanced techniques (e.g., same-origin policy, private browsing mode [7], [8] and cookie protection [26]) to manage the authenticators, the management in apps has to be constructed from scratch and often heavily relies on the underlying OS-level security mechanisms (e.g., permission-based access control, app-level sandbox) to mediate access to the authenticators. The problem is that the OS-level mechanisms merely take coarse-grained control which only keeps the authenticators within the app-level sandbox, but without necessary special treatment (e.g., origin-based isolation). Therefore, once the sandbox boundary is broken, the attacker is able to obtain the authenticators and take complete control of the authenticated sessions.

Backup is one of the most useful and desirable functionalities on Android. Although Android does not provide any official interface (e.g., APIs) for an app (i.e., the *backup app*) to backup the data belonging to other apps, developers have pioneered undocumented approaches to extricate themselves. So far, there have been two approaches to achieve this goal — one is via *rooting* the devices, through which a backup app can be granted the root privilege to access other apps' data; the other is to adopt the Android Debug Bridge (ADB)-based rooting-free alternative, which invokes a privileged *proxy* to conduct backup actions (detailed in Section II-B). In essence, backup implies violation of the underlying app-level sandbox mechanism. Therefore, the backup capability must be confined strictly. In practice, however, the developers of backup apps have not realized the risk of the backup channels and may unintentionally expose them to malicious apps. For example, the ADB proxy can be invoked without any access control [22]; backup data may be located to the publicly accessible storage. Consequently, installing a backup app may introduce threat of leaking authenticators for those web-based apps installed on the same device.

In this work, we investigate the (in)security of managing the backup capability by the backup apps and its impact on the security of authentication protocols. For the first step, we study the approaches of authenticator management used by contemporary web-based apps (detailed in Section II-A). Our study focuses on the app-side management of authenticators after they have been transmitted to mobile devices. We find that most apps locate the authenticators in their private storage, which relies on Android for access control. Consequently, the authenticators can be leaked to the attacks that compromise the backup channels. We define such attacks as *poaching attacks*.

With the knowledge of authenticator management in the apps, we then investigate how the authenticators can be leaked through the compromised backup channels. Our investigated subjects include all the backup apps that we can obtain from the Google Play market, including both *root-based* and *ADB-based* ones. We remark that we investigate only those apps that

backup other apps' data located in their proprietary folders, instead of those apps using Androids APIs to backup data like contacts, SMS and call history, because the latter are controlled by Android's label-based permission system and naturally do not have access to other apps' proprietary folders. We reveal how an app's authenticators are read, transmitted and positioned by those backup apps, and in turn, how they can be exposed to the attackers (Section III). As a concrete example showing the insecurity of the backup channels, we detail a comprehensive analysis on the state-of-the-art backup app named Helium (Section IV). Although this app has already employed various security mechanisms to prevent leakage, our study identifies four attack vectors through which the data belonging to benign apps can be leaked to delicately designed malicious apps (or *malware*). We further report three successful poaching attacks that exploit vulnerabilities including a severe logic flaw, uncontrolled access to the high-privilege proxy and exposure of backup data. These attacks completely compromise the confidentiality of the authenticators.

As a proof-of-concept, we design and implement an app named AuthSniffer which implements the poaching attacks to stealthily collect other apps' authenticators. AuthSniffer requests only permissions of network and SD card access, and transmits a small amount of data. We apply AuthSniffer to 117 real-world web-based apps which employs authentication schemes and find that 80 (68.4%) of them are subject to poaching attacks. As case studies, we detail concrete poaching attacks on popular and well-known apps, including Facebook, Facebook SSO SDK and Candy Crush (Section V).

Our findings should raise the alarm to the developers of both backup apps and web-based apps about this previously overlooked attack model. For the backup apps, the ADB proxy and the backup capability must be protected against unauthorized access. The logic in managing the ADB proxy and the communication protocol should be securely designed, and if possible, formally analyzed. For the web-based apps, security mechanisms should be actively conducted instead of treating the authenticators as normal data. The developers should avoid using the persistent authenticators, and should not implement the authenticator management from scratch, but instead, use Android's Account Manager.

Contribution. In summary, we make the following main contributions in this paper:

- **Raising Alert on a Previously Overlooked Threat Model to Authentication Protocols.** We for the first time investigate the security impact of the threat introduced by insecure backup channels on authentication protocols. Our study should raise the alert on this overlooked attack vector for the future design and analysis of authentication protocols on Android.
- **Practical Measurement on Impact of Poach Attacks.** We develop AuthSniffer to demonstrate how the malware can collect the authenticators through compromised backup channels. We conduct an evaluation on 117 real-world web-based apps and AuthSniffer successfully extracts authenticators from 80 of them. Our demo is posted online [3].
- **Recommendations of Securing Mobile Authentication Protocols.** We propose a number of effective recommendations for securing authenticator management on Android apps.

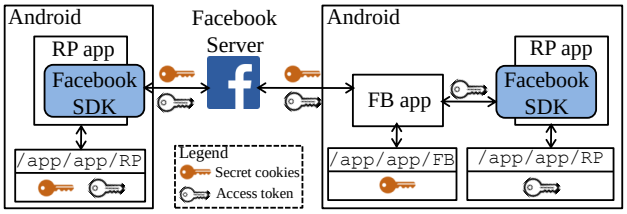


Fig. 1. Facebook SSO process and authenticator management with/without Facebook app

II. Preliminary Study and Background

A. Authentication Schemes for Web-based Apps

Android provides flexible options for the app developers to implement their authentication schemes. In order to learn the approaches that contemporary apps use, we manually study the top-ranked 100 apps downloaded from Google Play. In this section, we present our study.

Table I shows that the commonly used authentication schemes fall into three categories — basic authentication, Single Sign-on (SSO) and Account Manager based authentication. The former two usually reuse the interfaces designed for the browser-based clients, with customization and simplification to adapt for mobile devices, such as disabling CAPTCHA[1]. The latter is a unique feature supported by Android OS, and it provides a pluggable authentication scheme and automatic authenticator management for the apps.

Basic Authentication. The basic authentication stands for the traditional *knowledge* and *ownership*-based schemes. Most of the apps (34 out of 40) use the password (i.e., knowledge)-based scheme, while other apps (the remaining 6) employ phone number (i.e., ownership)-based one via sending a one time password through the SMS. In these implementations, the apps directly communicate with the authentication servers through the HTTP/HTTPS channel.

In essence, these apps play the role of the web browsers. For usability and convenience, most of them tend to reuse the authentication schemes designed for their browser-based counterparts. In most apps, the authenticators are also reused. For example, the login scheme of Facebook for mobile apps (through m.facebook.com) uses the same cookies as those for the browser clients (through www.facebook.com), although the UI has been simplified for the mobile devices. However, the problem arises after the authenticators are transmitted to the apps. Unlike their counterparts which can use the authenticator management provided by the well-evolved browsers, these apps have to implement their own management. We find that they use either the containers provided by Android, such as SQLite database and Shared Preferences [5], to manage the authenticators, or simply store them into a regular file. Eventually, the databases and files are all located in the app's proprietary directory (`/data/data/appname`).

SSO. SSO is an authentication protocol extensively employed by web applications. It allows the user to log into a web

[1]CAPTCHA is an acronym for Completely Automated Public Turing test to tell Computers and Humans Apart.

TABLE I. USAGE OF AUTHENTICATION SCHEMES ACROSS TOP-RANKED 100 APPS

W/O Authentication	Number	Schemes Used	Number[a]	Subcategory	Number
With	66	Basic Authentication	40	Activity-based	30
				Webview-based	4
				SMS verification	6
		SSO	40		
		Account Manager	16		
Without	34				

[a] The statistics in this column includes overlap, because some apps, e.g., Skype, may implement their own schemes while also embedding some SSO services.

application (named a *relying party* or RP) using her account registered with another web site (named an *identity provider* or IDP). To enable the user to use an SSO scheme on mobile platforms, the apps usually import the SDK provided by the IDP into their code.

The SDK may behave differently with or without the presence of the IDP's mobile client, and this leads to different approaches to managing the authenticators. Figure 1 takes the Facebook SSO as an example to demonstrate this difference. If the Facebook app has been installed (left-hand side box), the SDK invokes it to obtain the *access token* through Android's inter-process communication. The authentication process in turns happens between the Facebook app and the Facebook server. If no Facebook app installed (right-hand side box), the SDK invokes Webview to communicate with the server. The Webview is a browser-like widget, but it is different from the browser in terms of that it runs completely within the app's process space. We highlight this difference because these two approaches are fundamentally different in terms of security. In the former case, the authenticators of the IDP (i.e., the secret cookies) and the RP (i.e., the access token) are respectively located in their proprietary folders. IDP's authenticators are under its control. However, in the latter case, all authenticators including those belonging to the IDP are located in RP's folder. IDP's authenticators can be leaked if the RP is compromised, as shown by our case study in Section V-B.

Account Manager. Account Manager [1] is an Android service which provides a delegated authentication service and a centralized control of the user's web accounts and authenticators. The app can delegate the authentication and authenticator management to the Account Manager. The developers only need to implement some interfaces like `AbstractAccountAuthentication` and GUIs, and specify the authenticators that needs to be controlled by it. This service frees the app developers from the burdensome and error-prone process of constructing the authenticator management from scratch. The authenticators are finally stored in a database file named `accounts.db` in the `/data/system/users/0` folder (Android v4.4.2), which can only be read by the apps in the high-privileged user group (i.e., `system` and `root` group).

B. Backup Channels in Android

Backup is one of the most useful and desirable functionalities for Android users, since it enables the users to synchronize the apps' data[2] across devices, and can prevent loss of important data when device loss and system failures happen.

[2]The backup data we consider refer to the app's data located in its proprietary folder, instead of the user's data that can be accessed through Android's APIs, such as contacts, SMS and call history.

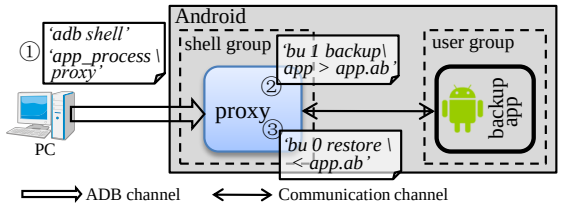


Fig. 2. Overview of ADB-based backup approach

However, there still lacks a straightforward way to implement this functionality. Android does not provide any API for the developers of the backup apps, and due to Android's app-level isolation, an app is not allowed to access other apps' data. Therefore, the developers have to push the boundary through undocumented ways. So far, there have been two approaches to implement backup—root-based and ADB-based approaches.

Root-based backup. The root-based approach requires to root the devices. After that, the backup app can be granted the root privilege such that it can access other apps' data located in their proprietary folders.

ADB-based backup. The ADB is a development tool which allows the app developers to communicate with the connected Android devices from their PC. It can be used to transmit files, invoke package manager (through the *pm* command) and issue shell commands. It also possesses part of the *signature-level* and *signatureOrSystem-level* permissions which only granted to the apps built into the system image and those signed by the vendor of the system image.

Figure 2 shows the overall architecture and process of the ADB-based backup approach. From the PC side, the user is required to open a shell using command "`adb shell`" and activate a *proxy* using `app_process`, a command used to execute a Java program (①) . The proxy process thus belongs to the `shell` group and inherits the permissions ADB possesses. Whenever the backup app needs to backup (②) data belonging to other apps and restore (③) backup data, it invokes the proxy to conduct the actions on its behalf. The communication channel between the proxy and the backup app can be either a local socket or a network socket. The proxy invokes Android's backup manager via commands "`bu 1 backup`" and "`bu 0 restore`" for backup and restore, respectively. The backup data is encoded as a file with `ab` extension.

III. OVERVIEW OF SECURITY ANALYSIS ON BACKUP APPS

A. Methodology and Threat Model

Since the backup data can be leaked at generation, transmission and storage, we focus on the following three key aspects to investigate how the backup data can be leaked.

TABLE II. SUMMARY OF THE ANALYSIS RESULTS ON THE EXISTING BACKUP APPS

Column 4: *replicate* means the backup apps directly replicate `/data/data/app2backup` folder, and *ADB* means through ADB proxy.
Column 5: S: SD Card, C: Cloud and P: PC connected through Web sockets.

Category	Apps	Installs	Approach	Destination	Encrypted?	Leak?
Root-based	My Backup	1,000,000 - 5,000,000	replicate	S, C	no	yes
	Ultimate Backup	500,000 - 1,000,000	replicate	S, C	no	yes
	Ease Backup	100,000 - 500,000	replicate	S	no	yes
	Titanium Backup	10,000,000 - 50,000,000	replicate	S, C	no	yes
ADB-based	Helium	1,000,000 - 5,000,000	ADB	S, C, PC	no	yes

- **Generation.** Which modules or code snippets in a backup app conduct the backup action?
- **Transmission.** Through which channels in a backup app are the backup data transmitted?
- **Storage.** At what places the backup data are located by a backup app?

The challenge to our analysis is that the implementation of the backup apps is too complicated to be completely analyzed (an intermediate-level app out of our subjects contains more than 500K lines of `smali` code). To cope with this, we use a top-down paradigm in our analysis. At the top level, we understand the overall architecture and working process of a given app, and identify its key modules (e.g., the ADB proxy, the communication channels/protocols and the backup interfaces). After identifying the key modules, we conduct an in-depth analysis on their internals for security properties.

In particular, we use a hybrid analysis which includes a whitebox analysis on the bytecode and a blackbox analysis on the communication messages. During the whitebox analysis, we decode the bytecode using the apktool [2] and extract the program dependence graph (PDG) using soot [6]. This step enables us to understand the app's behavior model (including the internal behaviors of its modules and the interactions among them) in processing the backup privilege. In the blackbox analysis, we capture the communication messages going through the network channel using Fiddler [4] and infer the communication protocol from them.

Threat Model and Assumptions. In our threat model, an attacker is able to trick the user into installing his malicious apps or inject malicious code into installed benign apps. The attacker can also disguise the malicious apps into other benign ones, such that they can use any package name as the attacker intends to. The malicious apps need to be granted only `INTERNET` and `READ_EXTERNAL_STORAGE` (to read SD card) permissions that are extensively requested by real-world apps.

Although we believe there are other sensitive information that can be leaked through the compromised backup channels, in this paper we focus on the authenticators. Suppose the victim logs into her account through a web-based app at time T_0. The goal of the attacker is to collect authenticators after T_0 and sends them out of the devices (so the malicious app needs at least `INTERNET` permission). In order to prevent the malicious app from raising the user's attention, the attacker can conduct a prior offline analysis on the targeting web-based app to learn the localization of the authenticators, such that the malicious app can extract them from the backup data, rather than send all the backup data through the network. We assume the cryptographic algorithms are correct and ideal, so that the backup data become secure once they are encrypted.

B. Summary of Analysis on Existing Backup Apps

We have analyzed all the backup apps that can backup other app's data from the Google Play market, and Table II summarizes our results. All the root-based apps simply replicate the private folders of other apps for backup, after they are granted the root privilege. A common problem is that all the backup apps locate the backup data in plain text into the SD card without any protection and access control. However, the SD card is publicly readable to the apps that are granted the `READ_EXTERNAL_STORAGE` permission[3].

It is more complicated to implement ADB-based backup than root-based backup, because the former needs to interact with a standalone proxy, while the latter only needs to invoke a "*cp*" command. We did not find logic flaws in the root-based apps except the problem of "un-encrypted backup data". In contrast, we found several vulnerabilities leading to data leakage in the ADB-based app which is implemented by adept developers (discussed shortly).

IV. POACHING ATTACK ON HELIUM: A DEMONSTRATION

Helium [12] is the only observed ADB-based backup app in stock. It is reported as one of the best apps in 2013 [27]. There exist channels in Helium which leads to the insecurity of the backup channels, even though it is developed by adept ClockworkMod who has released 19 apps on Google Play and accumulated more than 15 million installs. In this section, we detail our analysis and possible attacks on this app.

A. Revealing Internals of Helium

Figure 3 presents the overall architecture and working process of Helium, which are obtained through our manual analysis. In this section, we briefly explain how Helium processes its backup capability. To ease the understanding, we also list part of its control flow graph (CFG) in Figure 5.

The ADB proxy is named `ShellRunner`, which runs within a separated process from the *main app*[4] of Helium and belongs to the `shell` user group. After it is started through ADB, it creates an Android local socket server (N1 in Figure 5(a)). We denote this server with `lsserver`. Then it

[3]Beginning with Android 4.4, an app can have its private folder in the SD card, but none of the subjects locates the backup data to the private folder.

[4]We define all the modules of a backup app except the proxy as the main app.

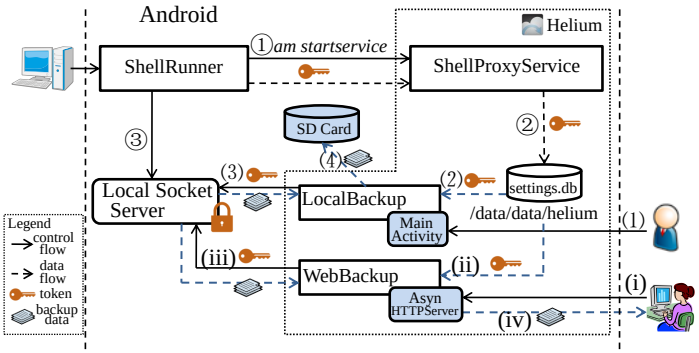
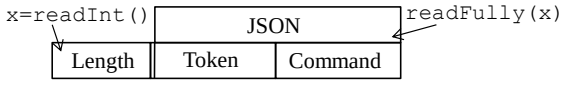


Fig. 3. Overview of Helium (obtained by reverse engineering)



`ShellRunner` first reads the length of the packet with `readInt()`, and then reads the token and the command.

Fig. 4. Structure of a command packet

generates a token (N3-N4 in Figure 5(a)) and starts a Service component named `ShellProxyService` with an explicit intent (using the `uri` of the Service as the target parameter) using the following command (step ① in Figure 3, N5 and N1'-N3' in Figure 5).

```
am startservice -n ShellProxyService -e
password "token"
```

As can be seen, the token is included in the `extras` bundle of the intent. The token does not expire with the whole lifetime of the proxy until the proxy is restarted. After receiving the intent from the proxy, `ShellProxyService` extracts the token and stores it into a database (named `settings.db`) located in Helium's proprietary folder, such that the main app of Helium can read it (step ②). Afterward, `ShellRunner` listens on `lsserver` for the requests from the main app (step ③ in Figure 3 and N6-N7 in Figure 5(a)).

Helium provides two interfaces for the user to conduct backup, which are implemented respectively by two modules. One is the local backup module, which uses an Activity to interact with the user and locates the backup data to the SD card (step (4)). The other is the web backup module, which creates a web server on the device and allows the user to connect it via a standard URL "`http://deviceIP:5000`". Through the web backup module, the backup data are transmitted out of the device (step (iv)). The communication protocol used to invoke services on the HTTP server is listed in Table III.

Eventually, both modules have to request `ShellRunner` to complete the backup actions. As aforementioned, `ShellRunner` is protected by the token. Therefore, both modules read the token from `setting.db` and use it as the credential to invoke `ShellRunner`. In order to understand the protocol used to invoke `ShellRunner`, we analyze the code snippet which processes the local socket communication. Figure 4 shows the identified format of the command packet sent to `ShellRunner`.

B. Overview of Helium's Security Properties

The above analysis demonstrates that Helium's developer has realized the significance of protecting the high-privileged proxy against unauthorized access from other apps. The token-based authentication is the most important mechanism to achieve this goal. Instead of using a hard-coded password, `ShellRunner` dynamically generates an token using Android's UUID generator, which guarantee it to be unique and unpredictable. The token is transmitted to the main app of Helium with an explicit intent (with an explicitly specified receiver). Unlike the implicit intents which are broadcast by Android OS, the explicit intent is directly sent to the specified receiver, such that the token cannot be leaked in this step. The token is located within Helium's proprietary folder, so it is under protection of Android's app-level isolation and access control mechanism.

Despite of the employed security mechanisms, Helium bypasses the user's authorization for the sake of usability. In particular, before actually conducting the backup/restore actions, Android displays a confirmation dialog for the user's authorization. Through this dialog, the user authorizes the backup action and inputs a password to protect the backup data. However, instead of prompting the user, the proxy directly injects a click event to remove the prompt.

C. Vulnerabilities and Attacks

Although the high-privileged proxy is seemingly well protected in Helium, we find that there exists mismanagement and logic flaws which may lead to leakage of the backup data and privilege of the proxy. In this section, we discuss these vulnerabilities and our poaching attacks on them. Unless otherwise stated, the attack app is denoted by AuthSniffer.

Vulnerability #1: Inconsistency of Lifecycle Management

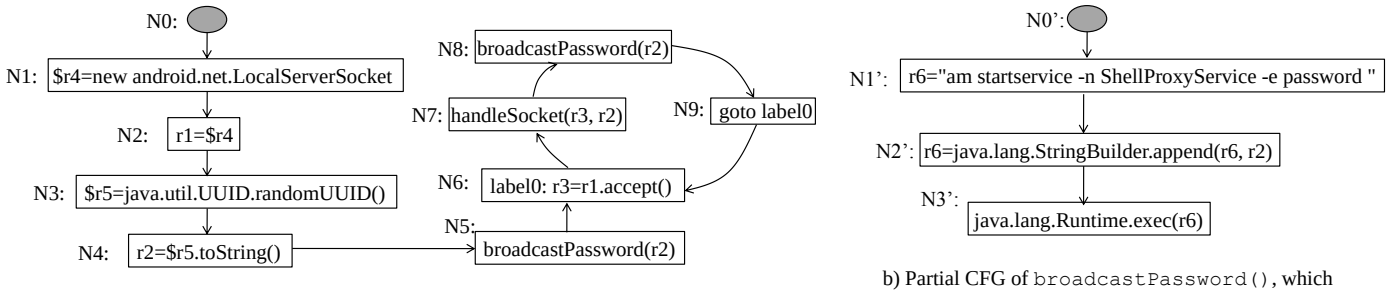
As discussed before, `ShellRunner` runs as an isolated process from the main app. From the perspective of Android OS, it is not a part of Helium, which implies that its lifecycle can be completely inconsistent with the main app. Therefore, after the user uninstalls the main app from the device, the remaining `ShellRunner` still keeps on running. This inconsistency itself seems not dangerous, but it may give the attacker a chance to exploit the privileged `ShellRunner`. In the following, we show that by combining with a logic flaw in the communication protocol between `ShellRunner` and the main app, the attacker can steal the token.

The logic flaw is caused by an infinite loop in the communication protocol between the `ShellRunner` and the main app. As can be seen from Figure 5, there exists a infinite loop, namely N6 → N7 → N8 → N9 → N6, such that `ShellRunner` never terminates its execution. This cause a problem that whenever an exception occurs in the communication in N7:`handleSocket`, `ShellRunner` attempts to restart the `ShellProxyService` in N8:`broadcastPassword` (N0'-N3' in Figure 5(b)).

Attacks (A1). The attacker can exploit this vulnerability to obtain the token. We consider an attack (denoted by A1) whose process is shown in Figure 6. The attacker installs a malicious app (i.e., AuthSniffer) to monitor the status of Helium by registering a broadcast receiver which filters the

TABLE III. PROTOCOL FOR INTERACTING WITH WEB BACKUP INTERFACE

URL	Method	HTTP Body	Description
`http://IP:5000/api/package`	GET	NULL	Fetch the list of installed packages
`http://IP:5000/api/backup.zip`	POST	Name of app to backup	Backup
`http://IP:5000/api/restore.zip`	POST	Backup data	Restore



a) Partial CFG of `main()` of `ShellRunner`

b) Partial CFG of `broadcastPassword()`, which starts `ShellProxyService`

Fig. 5. CFG of `ShellRunner`

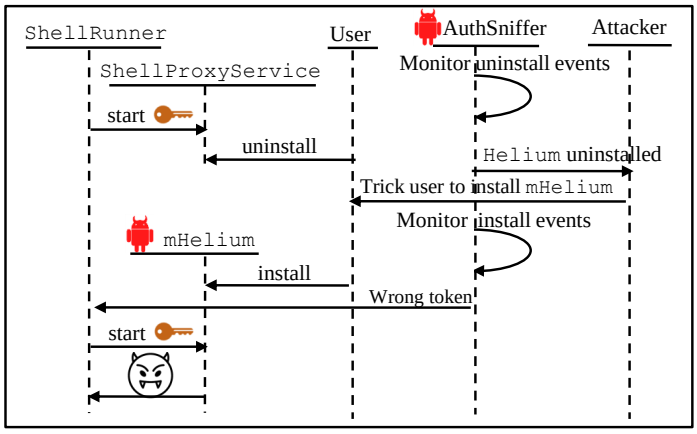


Fig. 6. Attack process of A1

`ACTION_PACKAGE_REMOVED` events. After the user has removed the original Helium, AuthSniffer notifies the attacker. The attacker then tricks the user into installing another malicious app (denoted by `mHelium`) that has the same identity as the uninstalled component `ShellProxyService`. Once detecting that `mHelium` is installed (similarly, via filtering events `ACTION_PACKAGE_INSTALL` and `ACTION_PACKAGE_ADDED`), AuthSniffer sends a command packet with a random password to `ShellRunner`, which causes an exception in `handleSocket` method (node N7 in Figure 5). `ShellRunner` then exits from `handleSocket` and enters `broadcastPassword`, where it tries to restart `ShellProxyService` using the intent including the token, but it actually sends the token to `mHelium`. Filtering the install/uninstall events does not require any permission. Therefore, this attack only needs the `INTERNET` permission.

Vulnerability #2: Unsanitized Command Execution

There exists another vulnerability which aggravates the security problem when A1 succeeds. In particular, `ShellRunner` does not encapsulate the backup/restore commands, but instead directly takes the received strings as commands to execute. This is similar to the vulnerability of unverified user inputs on the Web which leads to the notorious SQL injection and Cross-site Scripting (XSS) attacks. Therefore, once the attacker obtains the token using A1, he is able to request `ShellRunner` to execute any command that the high-privileged ADB proxy can execute. Most of the commands are security-sensitive, such as installing/uninstalling apps and injecting UI events. We list several of them in Table V of Appendix.

Vulnerability #3: Unprotected Web Interface

Although the web interface of Helium is designed for the access from the web browsers, any app granted the `INTERNET` permission is able to connect to it. Two problems make the web interface vulnerable. First, there is no access control to protect it. Therefore, the malicious app can invoke it using HTTP requests. Second, the web server is implemented as a Service, such that it keeps on running even when Helium is not in foreground. This increases the possibility of being invoked by the malicious apps.

Attack (A2). To conduct the attack, AuthSniffer can periodically scan the TCP port 5000 until it detects that Helium is listening to the port. It then sends an HTTP GET request to http://localhost:5000/api/package to fetch the list of the installed apps. If it finds the target app has been installed, it sends a POST request containing the name of the target to http://localhost:5000/api/backup.zip to invoke the web interface for the backup data of the target app.

Vulnerability #4: Uncontrolled Data Storage

As listed in Table II, Helium's local backup module is also subject to poaching attacks. The vulnerability is caused by two problems. First, as discussed in Section IV-B, Helium disables encryption on the backup data. Second, the local backup module locates the backup data into the SD card without any protection and access control, as shown in Figure 3. However, the SD card is publicly readable to the apps that are granted the `READ_EXTERNAL_STORAGE` permission.

Attack (A3). The backup data are located in the

`/sdcard/carbon` folder, which can be directly read by AuthSniffer.

Timing to Conduct Poaching Attacks. Attacks A1 and A2 are active attacks, in terms of that AuthSniffer is able to control the timing when it should conduct the attacks. In these two attacks, AuthSniffer periodically executes the `ps` command to check whether the target app is running. If it is, AuthSniffer periodically collects the backup data until the expected authenticators are identified. Attack A3 is a passive attack, since AuthSniffer is not able to control the timing of backup actions. Only when the user performs backup after she logs into her account, AuthSniffer can obtain the authenticators.

D. Extent of Poaching Attacks on ADB-based Backup

The poaching attacks on Helium only affect those apps locating authenticators in their proprietary folders. Therefore, the apps delegating authentication to the Account Manager are immune to the attacks, because Account Manager locates the authenticators in the `/data/system/user/0` folder which is only accessible by the apps in the `system` user group. In addition, an app can deny to participate the backup process by setting `android:allowBackup` that is `true` by default to `false` in its manifest file. The proxy who invokes the backup manager for backup actions thus is not able to obtain the app's data.

V. Measurement and Case Studies

To measure the impact of the poaching attacks on real-world authentication implementations, we conduct the attacks on the real-world Android apps. In this section, we first explain the extent of poaching attacks and quantify the apps that are subject to the authenticator leakage by poaching attacks. Then we use some top-ranked and extensively-used apps as case studies to demonstrate the concrete attacks on them.

A. Measurement

The subjects we used in our measurement include two sets of apps. The first set (denoted by set I) includes the top-ranked 100 apps discussed in Section II-A. We find that the proportion of game apps in this set is higher than apps of other categories. Therefore, to avoid bias, we randomly choose 10 categories from Google Play and select the top-ranked 10 apps from each of them, which forms the other set (denoted by set II).

The approach we use to investigate whether an app is subject to the poaching attacks is a blackbox approach. We set up a proxy between the target app and the web server, where we capture the HTTP/HTPPS messages and identify the authenticators from them using differential fuzzing that we proposed in AuthScan [9]. Then we conduct poaching attacks to obtain the backup data of the target app. To confirm whether the authenticator is included in the backup data, we restore the backup data to a "clean" instance of the target app running on another device and check whether the session is still valid. If the session is recovered, meaning the app is subject to the poaching attacks, we further extract the authenticators from the app's backup data to investigate how the app manages them.

In our experiments on the 100 apps in set I, we find that 66 (66%) of them contain authentication implementations.

TABLE IV. Statistics in experiments on set II

Categories	w/o Auth	With auth		
		Infected	Uninfected	
			AM[a]	w/o backup[b]
APP_WALLPAPER	8	2	0	0
APP_WIDGETS	7	1	2	0
BOOKS_AND_REFERENCE	6	3	0	1
BUSINESS	7	3	0	0
COMICS	8	2	0	0
COMMUNICATION	1	4	5	0
EDUCATION	7	2	0	1
ENTERTAINMENT	0	10	0	0
GAME	2	7	0	1
HEALTH_AND_FITNESS	3	7	0	0
Overall	49	41	7	3

[a] AM stands for those apps using the Account Manager.
[b] w/o backup stands for those apps disallowing to be backuped.

Among these 66 apps, 39(59.0%) are subject to poaching attacks, while the remaining 27 are immune to the attacks, because 16 of them use Account Manager and other 11 specify not to be backuped in their manifest files. The measurement results on set II are listed in Table IV. Although use of authentication and the distribution of different authentication approaches are significantly different among categories, the overall statistics on set II is similar to that on set I. There are 51 apps employing authentication schemes. Among them, 41(80.4%) are subject to the attacks.

B. Case Studies

We conduct the poaching attacks on three authentication implementations to show how the real-world apps can be infected.

Case Study #1: Attack Facebook app

Facebook app is the Android client of Facebook, the popular online social networking service. Since the mobile client enables the users to connect Facebook in a more accessible manner than the web browser, it has attracted more than 500 million Android users [21] so far.

Identify Authenticators. Facebook app uses a password-based authentication. It sends an HTTP POST request containing the user id and password to the Facebook server. After verifying the password, the server replies an HTTP response which contains an authenticator named `access_token`, which is used as the credential in the subsequent requests, for example, posting a new post and fetching friends' posts.

In addition, when analyzing this response, we find that it contains a set of cookies which are never used afterwards. We note that the `domain` field for each of them is "`.facebook.com`", so we turn to the authentication process when logging in through the web browser to identify their functionality. We capture and analyze HTTP messages when logging through the web browser. We find that two cookies named `c_user` and `xs` are actually the credentials indicating the user's login state. We then use the cookies extracted from the backup data to construct a login request and send it to the server, and this allows us to log into the victim's account.

Extract Authenticators from Backup Data. We search the values of the three authenticators in Facebook app's proprietary

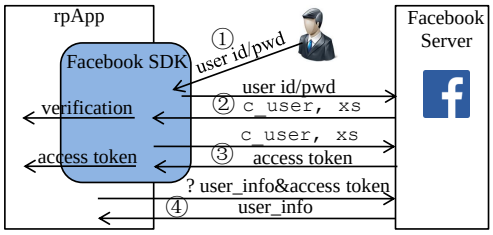


Fig. 7. Process of Facebook SSO

folder (i.e., `/data/data/com.facebook.katana`) and find that they are all located in a Sqlite database called `prefs_db`. The size of the database file is 112K, such that it may raise the user's attention if AuthSniffer directly sends the file out. To address this, we embed a Sqlite database manager in AuthSniffer to extract the textual information of the authenticators. In particular, the database contains three tables named `_shared_version`, `android_metadata` and `preferences`. The authenticators are stored in the table `preferences`. AuthSniffer invokes Sqlite to select all the text of the authenticators from the table, which reduces the size of data to send to only 88 bytes.

Case Study #2: Attack Facebook SSO SDK

Facebook has implemented the OAuth 2.0 protocol [17] as an SSO service. This service has been used by 26.3% of the top 100,000 websites [20]. Our investigation on the top-ranked 100 apps also shows that 29% of them embed the Facebook SSO service.

A typical Facebook SSO process is shown in Figure 7. Facebook SSO enables the users to login into a third-party website/app via authenticating themselves to Facebook. It also provides an authorization mechanism for the website/app to access the user's social data on Facebook using the *access token* as a credential.

Identify Authenticators. To use the Facebook SSO service, an app (denoted by *rpApp*) usually embeds the SDK provided by Facebook into their main app. As discussed in Section II-A, the SDK behaves differently with or without presence of the Facebook app. Since the authentication actually happens between the Facebook app and Facebook server when the Facebook app is installed, we consider the otherwise scenario in our case study.

Facebook SSO SDK initiates a Webview object which plays the role of the browser. Through the Webview, the SDK sends an HTTP POST request containing the user id and password to the Facebook server for authentication use. Facebook server then replies a response including the two cookies `c_user` and `xs` as authenticators (①-②). They are then used as credentials to fetch the access token (③). These two cookies are the same as those we find in our Facebook app case study, which indicate the user's login session to Facebook.

Extract Authenticators from Backup Data. Since `c_user` and `xs` are managed by the Webview, we analyze how the Webview handles the web cookies. As discussed in Section II-A, the Webview runs completely within the sandbox of rpApp. Therefore, its private data are also located in rpApp's proprietary folder. We find that by default, it locates its data into a subfolder named `app_webview` in the rpApp's folder. For the storage of the web cookies, the Webview uses a Sqlite database named `Cookies` located in the `app_webview` to store them. Therefore, as if rpApp allows its data to be backuped, AuthSniffer is able to obtain these two cookies from the backup data. Using text extraction, AuthSniffer reduces size of the transmitted data to 1.1K.

We discussed with Facebook security department about the way that the SDK handles the authenticators. We learnt that in the original design, the rpApp is regarded as part of the TCB. However, our case study shows that this design is obviously insecure. The management of the authenticators that are critical for Facebook is delegated to a third-party app developer who may not realize this duty at all. In [29], we further demonstrate how this violates the authorization property of Facebook SSO.

Case Study #3: Attack a Game App

In the case study on Facebook SSO SDK, we have analyzed the authentication protocol in the SSO process. In order to investigate the (in)security in managing the access token after the authentication, we analyze a concrete app which uses Facebook SSO. The app is a game named Candy Crush Saga [19], which is one of the most popular Android games and has been used by more than 100 million Android users.

Identify Authenticators. We use the same method as that in attacking Facebook app to identify the access token. The access token can be used to fetch the user's social information. For example, the following HTTP request can fetch the user's friend list.

```
GET https://graph.facebook.com/fql?access_token=CAACZCwCG
    ...&... HTTP/1.1
```

In this case study, we investigate how Candy Crush Saga manages the access token after obtaining it from Facebook. We analyze its bytecode and find that it uses Android's Shared Preferences to manage the access token.

Extract Authenticators from Backup Data. By default, the Shared Preferences creates several `xml` files in the folder `shared_prefs` located in rpApp's private folder as the permanent storage. After examining this folder, we find that two `xml` files respectively named `TOKEN_STORE_KEY.xml` and `DEFAULT_KEY.xml` include the access token. These files are quite small (1.1K and 301B respectively), so AuthSniffer directly sends them out of the device.

VI. Mitigation and Recommendations

Our study should raise awareness of developers of both normal apps and backup apps. In this section, we present recommendations to them, with the aim of preventing authenticators from leaking through backup channels.

A. Build Secure ADB-based Backup

It may be not a secure way to implement backup through rooting a device, given that rooting a device may introduce severe security risks [10]. In contrast, ADB-based approach seems a trade-off between usability and security. Based on our investigation on Helium, we present recommendations on how to construct an relatively secure ADB-based backup scheme.

Prevent Backup Privilege from Exposure. As discussed in Section IV-B, the token-based authentication is effective in mediating the access from the local socket. This mechanism can be employed to protect the web interface. We provide a preliminary design as following. Before starting the server socket, the backup app generates an token using the API `randomUUID()`. The token is then written into the `settings.db` as what `ShellProxyService` does. The app provides an Activity interface to display the token to the user. Whenever the user requests to backup from the PC, she reads the token from the Activity and uses it as the credential to connect to the web server.

In addition, the backup data must be encrypted before located to the SD card. The backup app should not automatically clear the prompt dialog (Secion IV-B), but should request the user to input a password.

Follow the Principle of Least Privilege. Besides the backup actions, the ADB proxy is capable of executing many security-sensitive commands. As a lesson learnt from the threat resulting from the vulnerability #2, these commands should not be executable by the backup app. In other words, the proxy should encapsulate them and provide the minimal set of APIs for the backup app to implement the backup/restore functionality. For example, in the Helium case, only two APIs `byte[] backup(string apkname)` and `restore(byte[] abfile)` are necessary.

Manage Lifecycle of ADB Proxy. The attack A1 exploits the logic flaw in managing the lifecycle of the proxy, which should raise an alert to the backup app developers. The principle is that the lifecycle of the proxy must be kept consistent with the main app. Whenever the app is uninstalled, the proxy should be terminated. After fixing the logic flaw in `ShellRunner`, Helium's approach is able to achieve this. In particular, whenever an exception occurs in the local socket communication (N7 in Figure 5), the proxy should not attempt to resend the token to the main app (N8). Instead, it should exit, such that `mHelium` is not able to obtain the token.

B. Protect Authenticators

To protect the authentication protocols of the web-based apps, their developers are also involved. We propose the following two recommendations for protecting confidentiality of the authenticators.

Avoid Persistent Authenticators. One possible countermeasure is to avoid using persistent authenticators[5]. The authenticators can be stored in the memory and are cleared when the app exits. This approach is similar to the private browsing mode in the modern browsers [7], [8]. The weakness of this solution is that it may increase the frequency of prompting the users for authentication.

Avoid Implementing own Authenticator Management. As discussed in Section II-A, Android's Account Manager is immune to the poaching attacks since the authenticator containers are located in the system's folder and only are accessible by apps in `system` group. In addition, it frees the developers from the error-prone process of constructing the authenticator management. Therefore, the most secure and advisable authenticator management for the web-based apps to use is the Account Manager.

[5]Persistent authenticators stand for the authenticators that are stored in a file.

C. Redesign the Backup Interfaces

Given the backup is a desired functionality, one satisfying strategy that Android designers can consider is to provide official APIs for this functionality. The APIs can be protected using permissions like `BACKUP`. However, note that this strategy remains debatable since it completely contradicts with Android's app-level sandbox paradigm.

Another feasible strategy Android can take is a *fine-grained manifest*. It should extend the `android:allowBackup` to allow the developers to specify the sub-folders that are not allowed to be backuped. The developers thus are able to locate the authenticators into these folders. A preliminary design for this we provide is as following.

```
<android:allowBackup="true", android:notallowfolders="
    folderA:folderB/folderC">
```

VII. Related Work

ADB Privilege Leakage. The problem of ADB privilege leakage is firstly discovered by Lin et al. [22]. In this work, the target is the screenshot apps which introduce the ADB proxies for programmatic screenshot. They develop an app called Screenmilker which extracts confidential information from the screenshots taken by invoking the unprotected ADB proxies. Screenmiker exploits the proxies that are without any access control and focuses on the approaches of extracting the information from the screenshots, while AuthSniffer focuses on exploiting the mismanagement and flaws in the backup app and extracting authenticators from the backup data. Both work raises the attention on the protection of ADB privileges.

Data/Capability Leakage in Android Apps. Two categories of advanced attacks have attempts to stealthily exfiltrate other apps' permissions and collect the user's sensitive information on Android. The first type of attacks are conducted by the so-called *sensory malware* that extracts and infers sensitive information from the data collected by the sensors on Android devices. For example, Soundcomber [25] attempts to extract the PIN numbers and credit card information from the audio records of phone calls. TouchLogger [11] tries to infer keystrokes on the touch screen based on the motion of the devices. The other type is via invoking the capabilities exposed (either deliberately or indeliberately) by other apps. Some apps deliberately perform a task requiring particular permissions for other apps without such permissions, which are called permission re-delegation attacks (or confused duty problem) [15], [14], [13]. Some apps fail to manage the interfaces to access their capability and data [23], [30], [16], such that the malicious apps can invoke the unprotected interfaces to obtain the previously un-granted permissions and access the victim app's data.

Security of Web-based Android Apps. The web interface of Android turns out to be insecure. Wang et al. [28] present the cross-origin attacks on the web-based apps. They reveal the leakage of the Facebook access token due to the vulnerable registration mechanism of "scheme" channels. Luo et al. [24] present the attacks targeting the weakened (compared with the web browsers) security features in the Webview.

VIII. Conclusion

We conduct the first investigation about the security impact of poaching attacks on the implementations of the authentication protocols in Android apps. We reveal that although security mechanisms has been employed, the backup privilege can be leaked due to the mismanagement and logic flaws. Our measurement shows an astonishing result that more than half of the web-based apps that have implemented authentication protocols are subject to the poaching attacks. We envision that our work would arouse awareness of the app developers about the fundamental differences in implementing authentication in Android clients and in browser-based clients. We also hope this work can raise the attention to this previously overlooked threat model when designing and implementing an authentication protocols.

Acknowledgement

This research is partially supported by the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-030).

References

[1] AccountManager. http://developer.android.com/reference/android/accounts/AccountManager.html.

[2] Android-apktool. http://code.google.com/p/android-apktool/.

[3] Demo of Poaching Attack. http://www.comp.nus.edu.sg/~a0091939/poaching/index.html.

[4] Fiddler. http://www.telerik.com/fiddler.

[5] SharedPreferences. http://developer.android.com/reference/android/content/SharedPreferences.html.

[6] Soot: a Java Optimization Framework. http://www.sable.mcgill.ca/soot/.

[7] Supporting Per-window Private Browsing. https://developer.mozilla.org/EN/docs/Supporting_per-window_private_browsing.

[8] G. Aggarwal, E. Bursztein, C. Jackson, and D. Boneh. An Analysis of Private Browsing Modes in Modern Browsers. In *Proceedings of the 19th USENIX Conference on Security (USENIX Security)*, 2010.

[9] G. Bai, J. Lei, G. Meng, S. S. Venkatraman, P. Saxena, J. Sun, Y. Liu, and J. S. Dong. AuthScan: Automatic Extraction of Web Authentication Protocols from Implementations. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS)*, 2013.

[10] BullGuard. The Risks of Rooting Your Android Phone. http://www.bullguard.com/bullguard-security-center/mobile-security/mobile-threats/android-rooting-risks.aspx.

[11] L. Cai and H. Chen. TouchLogger: Inferring Keystrokes on Touch Screen from Smartphone Motion. In *Proceedings of the 6th USENIX Conference on Hot Topics in Security (HotSec)*, 2011.

[12] ClockworkMod. Helium - App Sync and Backup. https://play.google.com/store/apps/details?id=com.koushikdutta.backup.

[13] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege Escalation Attacks on Android. In *Proceedings of the 13th International Conference on Information Security (ISC)*, 2011.

[14] W. Enck, M. Ongtang, and P. Mcdaniel. Mitigating Android Software Misuse Before It Happens. Technical Report NAS-TR-0094-2008, Pennsylvania State University, 2008.

[15] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission Re-delegation: Attacks and Defenses. In *Proceedings of the 20th USENIX Conference on Security (USENIX Security)*, 2011.

[16] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe Exposure Analysis of Mobile In-app Advertisements. In *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks (WISEC)*, pages 101–112, 2012.

[17] D. Hardt. The OAuth 2.0 authorization framework. RFC 6749, RFC Editor, Oct. 2012.

TABLE V. Security-sensitive commands ADB proxy can execute

Commands	Description
bugreport	Print dumpsys, dumpstate and logcat data
install/uninstall	Install/uninstall apks
forward	Forward requests to a specific port
broadcast	Broadcast an intent
set-debug-app	Set a particular app to debug mode
grant	Grant apps particular permissions
monkey	Inject interaction events

[18] H. Kelly. Facebook Mobile Users Surpass Desktop Users for First Time. http://www.cnn.com/2013/01/30/tech/social-media/facebook-mobile-users/.

[19] King.com. Candy Crush Saga. https://play.google.com/store/apps/details?id=com.king.candycrushsaga.

[20] Leadledger. Facebook Connect. http://www.leadledger.com/tech/Facebook-Connect.

[21] C. Lee. Facebook Mobile User Stats: 192M on Android, 147M on iPhone. http://www.idownloadblog.com/2013/01/07/facebook-unveils-mobile-stats/.

[22] C.-C. Lin, H. Li, X. Zhou, and X. Wang. Screenmilker: How to Milk Your Android Screen for Secrets. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS)*, 2014.

[23] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS)*, pages 229–240, 2012.

[24] 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)*, pages 343–352, 2011.

[25] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In *Proceedings of the 18th Annual Network & Distributed System Security Symposium (NDSS)*, Feb. 2011.

[26] U. Shankar and C. Karlof. Doppelganger: Better Browser Privacy Without the Bother. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, pages 154–167, 2006.

[27] W. Shanklin. Best Android Apps of 2013. http://www.gizmag.com/best-android-apps-2013/30238/.

[28] R. Wang, L. Xing, X. Wang, and S. Chen. Unauthorized Origin Crossing on Mobile Platforms: Threats and Mitigation. In *Proceedings of the 2013 ACM Conference on Computer Communications Security (CCS)*, pages 635–646, 2013.

[29] Q. Ye, G. Bai, and J. S. Dong. Formal Analysis of A Single Sign-on Protocol Implementation for Android. In *Proceedings of the 20th International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2015.

[30] Y. Zhou and X. Jiang. Detecting Passive Content Leaks and Pollution in Android Applications. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS)*, 2013.

Appendix

The ADB proxy has the privilege to execute a set of security-sensitive commands, in Table V, we list several of them. We refer the interested readers to the documentation of the Android Debug Bridge (http://developer.android.com/tools/help/adb.html) for the full list.