

Article

Personal Information Leaks with Automatic Login in Mobile Social Network Services

Jongwon Choi, Haehyun Cho and Jeong Hyun Yi *

School of Computer Science and Engineering, Soongsil University, Seoul 156-743, Korea;
E-Mails: bluster100@gmail.com (J.C.); haehyuncho@gmail.com (H.C.)

* Author to whom correspondence should be addressed. E-Mail: jhyi@ssu.ac.kr;
Tel.: +82-2-820-0914; Fax: +82-2-823-0914.

Academic Editors: James Park and Wanlei Zhou

Received: 17 January 2015 / Accepted: 5 June 2015 / Published: 10 June 2015

Abstract: To log in to a mobile social network service (SNS) server, users must enter their ID and password to get through the authentication process. At that time, if the user sets up the automatic login option on the app, a sort of security token is created on the server based on the user's ID and password. This security token is called a credential. Because such credentials are convenient for users, they are utilized by most mobile SNS apps. However, the current state of credential management for the majority of Android SNS apps is very weak. This paper demonstrates the possibility of a credential cloning attack. Such attacks occur when an attacker extracts the credential from the victim's smart device and inserts it into their own smart device. Then, without knowing the victim's ID and password, the attacker can access the victim's account. This type of attack gives access to various pieces of personal information without authorization. Thus, in this paper, we analyze the vulnerabilities of the main Android-based SNS apps to credential cloning attacks, and examine the potential leakage of personal information that may result. We then introduce effective countermeasures to resolve these problems.

Keywords: credential; Android vulnerability; authentication; personal information leakage

1. Introduction

Because smart devices offer the ability to access services from anywhere, they often carry a great deal of personal data. This has the disadvantage that these personal data present an attractive target to hackers. Various mobile social network service (SNS) apps allow users to create a network to manage relationships among friends based on their accumulated contacts [1–5]. For example, the Google Account app, which is built-in to the Android system, makes personal information such as a user’s contacts, e-mail, call history, schedule, and web history accessible after the user has logged in. For users’ convenience, most mobile SNS apps provide an automatic login function using credentials, which are security tokens that certify a user’s identity. Credentials are generally used by the server to validate a requesting user. When a user logs in to the mobile SNS server, they enter their ID and password to get through the authentication process. From that point on, they can then be logged in without re-entering their ID and password. Nowadays, the majority of SNS service providers have adopted such credential-based authentication methods. Thus, as credentials are very sensitive data [6–8] that potentially allow users to access a full range of services, they should be carefully managed by mobile devices.

However, if the credentials are managed insecurely, they may be duplicated. Attackers can then use the cloned credentials to log in to the victim’s service account, and are able to steal a variety of the victim’s information [9–11]. We call this a credential cloning attack. In this paper, we explore the robustness of some well-known mobile SNS apps against credential cloning attacks, and then introduce countermeasures to improve security.

This paper is organized as follows. First, in Section 2, we introduce some background information. In Section 3, we describe a credential cloning attack in more detail. Section 4 presents experimental results for the majority of mobile SNS apps. In Section 5, we outline our proposed countermeasures to improve security in these apps. Finally, we give our conclusions in Section 6.

2. Background

2.1. Android Storage Partition

Android provides storage space to save credential data used by an app. Table 1 presents partition information for the Android file system. Among these, the /system and /data partitions are the most important. The /system partition is the storage space used by system apps installed by Google, device manufacturers, or network operators. The /data partition is where data from user-installed apps are stored.

Table 1. Partition information for Android storage.

Mount Point	Description
/cache	Storage location for temporary files
/system	Location of system apps
/data	Location of user-installed apps
/sdcard	Location accessible by external SD card

Credentials are stored on these partitions depending on the privilege of the apps. That is, the credentials of system apps are located on the /system partition, whereas credentials of user-installed apps are on the /data partition. Each app runs by reading and writing the credential file on the corresponding partition.

2.2. Credential Location and Application Programming Interface (API)

The credential locations of popular SNS apps are summarized in Table 2. All apps use the API provided by the Android SDK (Software Development Kit) related to credential management. App developers store various configuration data inside the smart devices. Credential data are a form of configuration file for automatic login, and are managed by the Android API. For more details, system-privileged apps such as Google Account and Samsung Account store their credential files in a database form. To access this data, Account Manager is invoked as an API. In the case of user-privileged apps, credential files are stored in XML or database form, and are accessed by invoking SharedPreferences or SQLite, respectively. With this knowledge about the locations of each app, an attacker is able to obtain credentials.

Table 2. Credential locations and related API for popular SNS apps.

Name	Location	API
Google Account	/data/system/accounts.db	AccountManager
Facebook	/data/data/com.facebook.katana /databases/prefs_db.db	SQLite
Kakaostory	/data/data/com.kakao.story /database/kakao_story.db	SQLite
Cyworld	/data/data/com.btb.minihompy /shared_prefs/SKCOMMS_ACCOUNTDATA.xml	SharedPreferences

2.3. Sandbox

The Android platform uses Linux's user-based protection feature, called sandbox [12], to differentiate applications, with each app assigned a UID (User Identifier). To support this feature, sandbox should be implemented in the kernel level. Thus, user-privileged apps cannot exchange data with other apps. In other words, because the credential is a kind of data, it is stored where it cannot be accessed by other apps. Thus, we apply a well-known rooting technique to enable the attacker to access the victim's credentials on the target device.

2.4. Unauthorized Data Access

In situations where an attacker cannot physically access the victim's smart device, a social engineering attack can be used to steal personal data. This can be achieved by installing malware on the smart device. A typical example is an application repackaging attack. Because of certain structural features of Android applications, repackaging attacks [13] can be applied without difficulty.

However, as we need to extract the credential from the victim's device, we assume that the attacker can access the smart device physically, and is able to activate ADB (Android Debug Bridge). ADB is

a software tool included in the Android SDK to manage Android devices and emulators. This method can be used when the smart device is secured with its screen locked (requiring a PIN, password, pattern lock, *etc.* to access), or when the smart device's screen is physically damaged. This is a very serious condition because the attacker can directly install an app or obtain administrator privileges, directly access the device's internal data, and then leak the information out of the smart device. The main features of ADB are device connection detection, shell access, data extraction/insertion (rooting device), and application installation/uninstallation. ADB's pull instruction is used to extract personal information.

In the case that ADB is inactive, a recovery mode attack can be applied. For this, the attacker must first make a customized recovery image. The attacker then loads this image onto the recovery partition using malicious code. After uploading the image, the attacker can raise their privilege level, access the user's personal data, and then leak the information.

2.5. Code Obfuscation

Code obfuscation [14] is a kind of program transformation that makes it difficult for attackers to read the program logic, thereby increasing the resistance to reverse engineering. There has been a lot of research on native code obfuscation in recent decades with the aim of protecting PE (Portable Executable) or ELF (Executable and Linkable Format) files. Nowadays, with mobile apps increasingly popular, issues related to Java code obfuscation have become an important topic of research. In practice, there are five major obfuscation features that can be applied in Java: renaming identifiers, control flow randomization, string encryption, API hiding, and class encryption. Simply put, renaming identifiers involves replacing the original names of the classes, fields, and methods with meaningless words. This increases the time required to reverse engineer the app structure. The objective of control flow randomization is to mislead the decompiler into adding some junk code, thus protecting the logic of the method and variables. In string encryption, the original string constant in Java is encrypted. API hiding is used to prevent a static analysis from discovering sensitive APIs, generally via a reflection mechanism. The mechanism for encrypting an entire class is called class encryption. When an app contains encrypted classes, it decrypts these first and loads them into a class pool on the virtual machine.

3. Credential Cloning Attacks

3.1. Assumptions

As mentioned in Section 2.4, to enable credential to be cloned, it is assumed that physical access to the device is possible, and that ADB is activated. In this scenario, even if the smart device is protected by a screen lock or has a physically damaged screen, the attack is still possible. In reality, to prevent various data leakage from lost devices, smart devices provide a certain user authentication mechanism of their own to protect against shoulder-surfing attacks [15,16], smudge attacks [17], and so on. In our scenario, we do not consider security based on such human–computer interaction to be available. It is assumed that the attacker can simply extract the credential data from the smart devices, insert the extracted credentials to the attacker's smart device, and access the target's various services.

3.2. Attack Scenario

The attacker starts rooting the smart device. This process is done to acquire administrator permission to access the data section. Rooting removes the permission constraints placed by the device provider, and can be performed using well-known rooting tools such as Odin3 [18] and Super One Click [19]. After rooting the smart device, the attacker connects a USB cable from a PC to the smart device, and then uses ADB's "pull" instruction to extract the credential files. After extracting the credentials stored on the victim's smart device, the attacker inserts them into their own smart device. By connecting the desktop to the attacker's smart device with the USB cable, the attacker can use ADB's "push" instruction to insert the victim's credential files. Typically, the insertion path is the application's data directory, which is `/data/data/[package_name]/`.

Figure 1 illustrates the process of credential cloning using an automatic login. After extracting the credential stored in the victim's smart device, the attacker inserts it into their own smart device. Then, when the app is running, it requests authentication from the server using the victim's credential, and the server starts the authentication using the victim's account. Using the victim's account, the attacker is able to automatically log in without entering the ID and password. In general, because credentials are constructed with the user's ID, password, and other personal information as an input, if an ID and password were stored in the form of plaintext, there would be a number of security problems. Therefore, app developers apply various techniques, including encryption, to protect such sensitive data before they are stored on smart devices. However, in a credential cloning attack, the attacker does not need to decrypt or analyze the credential data, because the data itself can be used for authentication from the server. Having completed the authentication process, the server transmits all of the victim's personal information to the attacker's smart device, and the attacker is free to analyze the data and gain access to the victim's information.

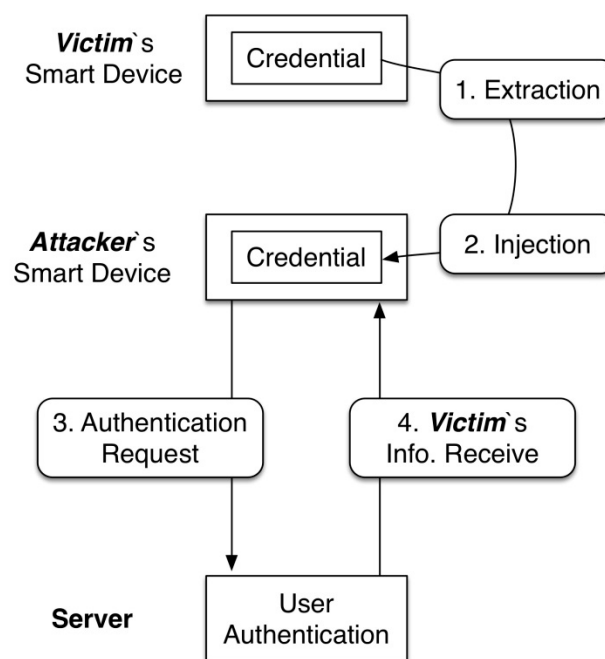


Figure 1. Overview of a credential cloning attack.

3.3. Attack Method

3.3.1. Analysis of Credential Location

An attacker can reverse engineer target apps such as Google Account and Facebook to determine the location of their credentials. After extracting the smali code [20] from a decompiled Google Account app, we can find the credential management routine from the source code, as shown in Figure 2. The package path is com/google/android/gsf/loginservice, and the class file name is GLSUser. A variety of information about the corresponding account is stored in the mAccount object. We can also infer that the e-mail address and encrypted password used during each login are stored in the mEmail and mEncryptedPasswrod string objects, respectively.

```
.class public Lcom/google/android/gsf/loginservice/GLSUser;
.super Ljava/lang/Object;
.source "GLSUser.java"

#instance fields
.field mAccount:Landroid/accounts/Account;
.field public mEmail:Ljava/lang/String;
.field mEncryptedPasswrod:Ljava/lang/String;
```

Figure 2. Routine to manage credentials for Google Account.

Figure 3 shows the routine used to store credentials in the Google Account app. We can see that the firstSave method is called as the app begins to run; the addAccountExplicitly method of the AccountManager API is invoked within this method. When the addAccountExplicitly method is invoked, the credential file corresponding to the unique database file of the actual smart device is stored. However, if the AccountManager API is used, the credential will be managed as accounts.db in the internal /data/system partition. All apps using that API will store their credentials in /data/system/accounts.db. The location of the credential for the Google Account app can be similarly found.

```
.method private firstSave(zLjava/lang/String;...)
.locals 13
.parameter "browserFlow"
.parameter "sid"
.parameter "lsid"
        .
        .
        .
iget-object v10, p0
        Lcom/google/android/gsf/loginservice/GLSUser;
        ->mAccount:Landroid/accounts/Account;

invoke-virtual {v9, v10, v8, v5},
        Landroid/accounts/AccountManager; ->addAccountExplicitly
        (Landroid/account/Account;Ljava/lang/String;
        Landroid/os/Bundle;)Z
```

Figure 3. Routine to store credentials for Google Account.

Figure 4 shows the routine for creating credentials for the Facebook app. As we can see, Facebook uses the SQLite API to store its credential. With this knowledge, we can figure out that the credential is managed in a database form, and that a credential file is created under the name `prefs_db`.

```
.method public constructor <init>(Landroid/...)V
.locals 3
.parameter

.prologue
.line 29
const-string v0, "prefs_db"

                                .
                                .
                                .

const/4 v2, 0x1

invoke-direct {p0,p1,v0,v1,v2}, Landroid/database/sqlite
    /SQLiteOpenHelper;-><init>(Landroid/content/...)V

.line 30
return-void
.end method
```

Figure 4. Routine to create credentials for Facebook.

3.3.2. Credential Extraction

Once the attacker has found the location of the credential and its relevant API, they root the smart device. The attacker can then extract the credential using the ADB “pull” command. A detailed description is as follows (see Figure 5).

- 1) Connect the rooted smart device to a desktop machine with a USB cable
- 2) Run the shell by entering the “adb shell” command in the prompt
- 3) Obtain permission to access “/data/” partition by entering the “su” command in the prompt
- 4) Change the prompt directory that includes the credential using “cd /data/system/<package_name>”
- 5) Change the credential file’s extraction permission using “chmod <mode> <file>”
- 6) After closing the shell, copy the credential from the smart device onto the desktop using “adb pull <remote> <local>”

```
C:\Google>adb shell
shell@android:/ $ su
su
root@android:/ # cd /data/system
cd /data/system
root@android:/data/system # chmod 777 accounts.db
chmod 777 accounts.db
root@android:/data/system # ll
ll
-rwxrwxrwx system system 334 SimCard.dat
-rwxrwxrwx system system 90112 accounts.db
                                .
                                .
                                .

C:\Google>adb pull /data/system/accounts.db
3833 KB/s (4192850 bytes in 1.068s)
```

Figure 5. Credential extraction procedure.

3.3.3. Credential Insertion

After extracting the credential from the victim's smart device, the attacker inserts it into their own smart device. The detailed procedure is as follows (see Figure 6).

- 1) Connect the attacker's smart device to the desktop with a USB cable
- 2) Copy the credential file extracted from the victim's smart device into the data directory of the attacker's smart device using "adb push <local> <remote>"
- 3) Launch the app in the attacker's smart device

```
C:\Google>adb push accounts.db /data/system/accounts.db
3833 KB/s (4192850 bytes in 1.070s)
```

Figure 6. Credential insertion procedure.

Currently, Android apps store credential files in the internal /data directory, and it is assumed that the attackers are always able to repackage the target apps according to their intentions. To protect against such a repackaging attack, many apps encrypt the credential file with a unique device identifier such as IMEI or ANDROID_ID to make the copied credential meaningless on any other devices. However, ANDROID_ID can be accessed by the attacker using the relevant API included in the Android SDK (see Figure 7). The ANDROID_ID of the victim's device can be determined from the smali code obtained by disassembling the bytecode corresponding to the API for finding ANDROID_ID. The attacker then uses the extracted ANDROID_ID for their malicious purpose. Figure 8 shows that the extracted ANDROID_ID can be inserted into the smali code. The original code used the v0 register to read ANDROID_ID dynamically from the device. However, the modified code statically sets the v0 register value to the victim's ANDROID_ID. After recompiling the modified smali codes, the victim's credential is successfully inserted, even though a device identity checking mechanism is applied.

```
Settings.Secure.getString(getContentResolver(), "android_id")
```

Figure 7. Source code for reading ANDROID_ID.

```
const-string v1, "android_id"
invoke-static {v0, v1}, Landroid/provider/Settings$Secure;
    ->getString(...)Ljava/lang/String;
move-result-object v0
const-string v0, "9b363ac21b566fc2"
    ### ANDROID_ID of Victim's Smart Device
if-nez v0, :cond_0
const-string v0, ""
:cond_0
invoke virtual {v0}, Ljava/lang/String;->hashCode()I
```


Figure 8. Smali code to which the victim's ANDROID_ID is inserted.

4. Experimental Results

In this section, we examine apps' vulnerability to information leakage from the credential cloning attack described above. We experiment with Google Account version 4.4.2-937116 and Facebook version 17.0.0.23.16 (note that, as of March, 2015, the latest version of Facebook is 20.0.0.25.15. and that credential cloning attack no longer works on this or later versions).

4.1. Google Account

The main objective of Google Account is to sync various data stored on the server and conveniently backup and restore data for users. If this function were to be used by an attacker who had automatically logged-in using a victim's credential, then Google Account could be used to gather all sorts of data from the breached account. Figure 9 shows the results of a successful automatic login from an attacker's device by copying a victim's Google Account credential.

Table: 

	_id	name	type	password
1	41	SolCalendar	net.daum.android.	1234
2	42	blu[REDACTED]@gmail.com	com.google	oauth2rt_1/xF4Tz4p'

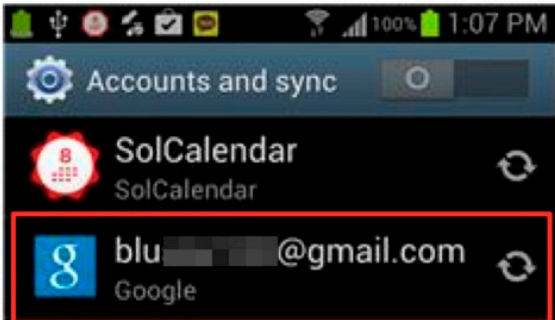


Figure 9. Result of an automatic login to Google Account using a victim's credential.

Figure 10 illustrates the leakage of contacts stored on the Google server. This may occur when an attacker synchronizes their own smart device using a victim's Google Account credential to automatically log-in. Because Google Account not only syncs contacts, but also user information such as e-mail, calendar, app data, web history, and so on, even personal information such as the victim's name, gender, age, and occupation is also leaked.

Google Account provides more than a simple data backup and recovery service. Google provides a variety of APIs for app developers, creating a development environment in which a diverse range of apps can use the Google Account credential. Consequently, with the Google Account credential, an attacker can access many services provided by numerous applications using a victim's account. Table 3 lists the most commonly used apps that are built to interact with Google Account.

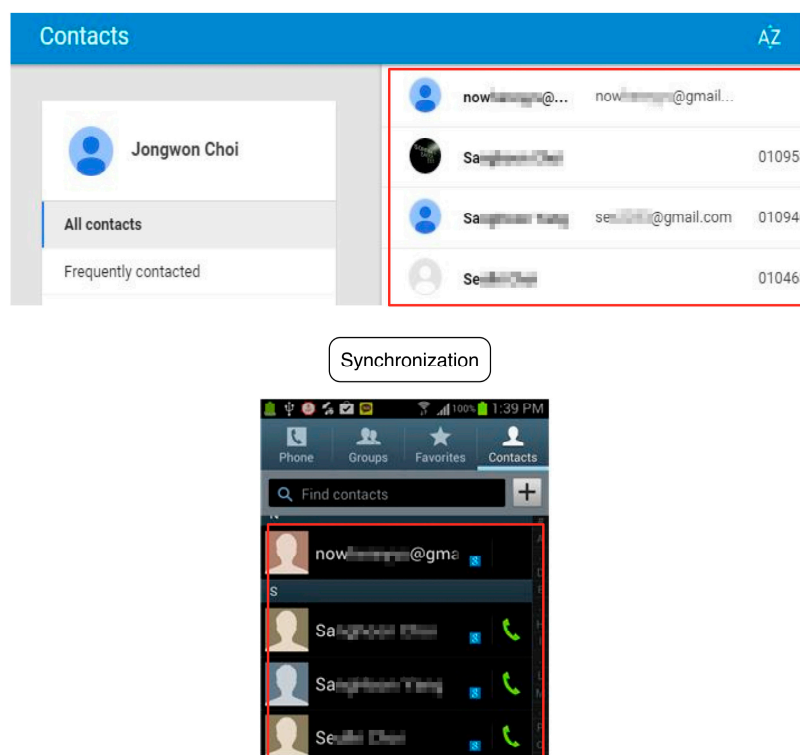


Figure 10. Contacts leaked by synchronization of Google Account.

Table 3. Selected apps built in Google Account.

Apps	Description	Leaked Information
Hangout	Chatting program	Friends and chat history
Chrome Browser	Web browser	Bookmarks, secret tabs
Google Calendar	Scheduling calendar program	Schedule, task lists
Google Drive	Document sharing	Privately shared documents
Google Map	Location information service provider	Location information
Dropbox	Photos and documents storage	Personal photos, documents, <i>etc.</i>

4.2. Facebook

SNS apps support connections between online users who share similar interests, and provide a variety of communication outlets such as social activities. Thus, if an attacker procures the credentials of an SNS app, they can obtain information on the victim's hobbies, interests, and networks. Figure 11 illustrates a successful automatic login by an attacker who has inserted the Facebook app credential into their own smart device. This confirms that by simply inserting the credential, an attacker is freely authenticated by the server, and can access the victim's account on the Facebook server.

From the services provided by Facebook, the attacker can gather personal information such as the victim's name, occupation, *etc.* Moreover, the attacker can access information about the victim's friends and groups from the leaked posts or messages on the victim's SNS. Table 4 shows the major SNS apps and the information that each is vulnerable to leaking.

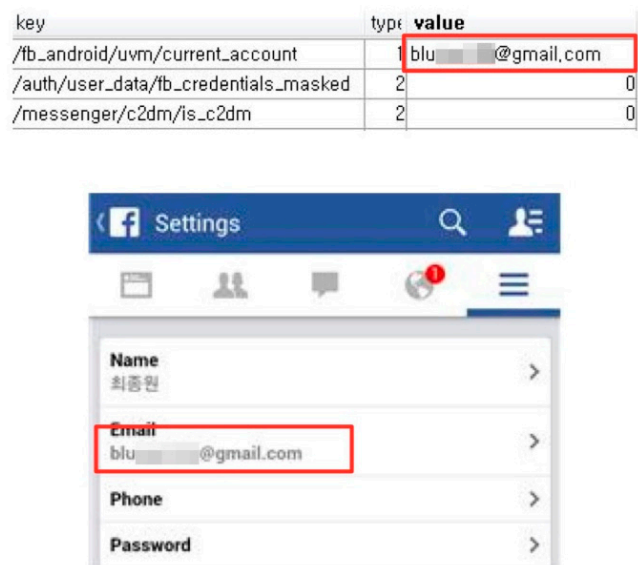


Figure 11. Result of an automatic login to Facebook using victim’s credential.

Table 4. Information leaked from SNS apps.

Type	Facebook	Kakaostory	Cyworld
Personal Information	O	O	O
Friends’ Information	O	O	O
Group Information	O	X	O
Posted Messages	O	O	O
Location Information	O	X	X

The real device’s /data directory contains more than just the credential itself. Figure 12 shows a Facebook database file opened by SQLite Database Browser. Inside the friend table in the database file, we can see the victim’s friend list and profile pictures. Thus, as well as the credential, the /data directory includes other files that contain sensitive data. The experimental results show that various data such as friend lists, address books, profile pictures, etc. could be leaked from these database files.

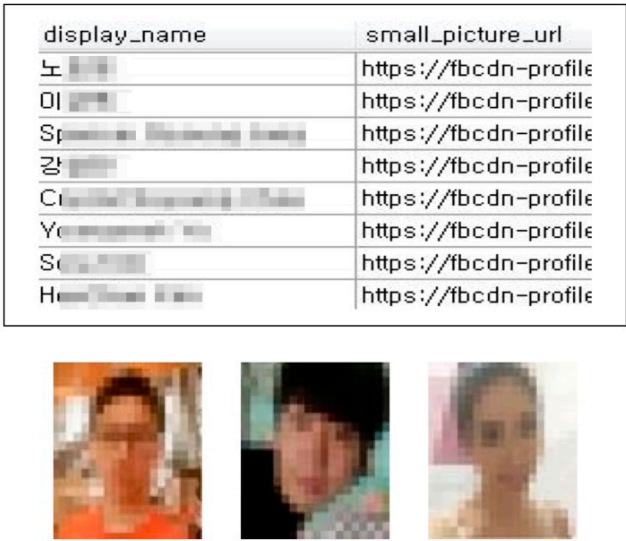


Figure 12. Additional information leaked from a Facebook database.

5. Countermeasures

In this section, we introduce several methods to mitigate the many threats caused by a credential cloning attack.

5.1. Server-Based Credential Management

Bear in mind that, as we have mentioned before, there is a clear limit to what client (*i.e.*, mobile device) based security techniques can do. The existing method sends credentials to the server. Therefore, if data are hijacked, they can be misused, assuming that any apps on the client side can be repackaged. Our observations indicate that the code for identification should not be stored on the client side. Although most apps employ very similar “lock” functions, whereby the password or pattern value is stored on the client side, all of the apps we tested could be hacked within minutes. Because it is doubtful whether lock functions are beneficial to an app’s performance (and because not all apps use the lock function), it is worth considering the implementation of an additional identification process. We thus suggest using additional data, other than the password, that the user knows but a random attacker does not. This method would have the user enter a PIN along with the credential whenever a smart device tries to automatically log in to an app after reading the credential file. By sending the PIN along with the credential, the server can check whether the owner of the credential has changed. Even if an attacker hijacks the credential file, they cannot access the victim’s account unless they also know the PIN. Although this method has the disadvantage of asking users for additional input, it could be viable, especially considering the current solution in which an external hardware device such as an OTP (One-Time Password) is considered to be acceptable.

5.2. Hardware-Assisted Credential Management

The major weakness in credential management for existing apps is that the credential is simply stored within the smart device without any security measures. To block credential cloning attacks, the credential should be stored separately and managed on an external device. In this way, an attacker would be unable to gain access to the credential using only the victim’s smart device. As a result, without the ID and password or credential, the attacker cannot login to the service provider, and cannot gain access to any of the victim’s information. This solution has the drawback of requiring a separate medium. However, as a countermeasure for recent mobile app security vulnerabilities, security solutions that utilize a variety of external devices (including wearable computing devices such as the smart watch) are being actively implemented. Thus, it is predicted that, gradually, the resistance to security methods that require a separate device to be carried will decrease.

5.3. Session Management

In the case of credential cloning, the most effective method of preventing the leakage of information is session management on the server. The current problem is that messages can be simultaneously sent and received in real time. It is important that the legitimate user device is identified on the server, and that only one session is maintained. Although most apps do not allow duplicate access from different devices, they do permit session requests that have the same device ID. Ironically, this results in

maintaining two or more connections per user. The very nature of mobile devices, however, will make it difficult to determine an adequate number of connections. The use of discrimination methods based on existing signaling systems should be considered, as well as implementing the additional identification method introduced in Section 5.1.

5.4. Code Integrity Check

In terms of server-based integrity checking, the most commonly used method is hash authentication of the signature. The hash value is useful for verifying the integrity of falsified areas of code that have been repackaged by an attacker. However, if this routine exists within the app's internal code, the attacker can also tamper with this method, making it insecure. For this reason, this signature value must be sent to the server without the client internally verifying the signature. Because the transmitted data may be exposed online, SSL communication must be used. To prevent attacks on memory dumps, the memory area containing the hash string should be immediately deallocated.

A second method only provides necessary information to the registered client if the initial verification is successful. Currently, most apps run the integrity checking routine once, and allow the app to run normally if the return value from this check is correct. This method is very vulnerable to reverse engineering, and leaves an identification flag within the server. This method of sending information should only be used when this flag is normal. Using this method, even if an attacker were to repackage the app to make it appear as a normal app, no information can be gained from the server. For this to happen on an actual server, more elaborate verification is needed. We must keep in mind that, because apps linking the dynamic library file can be falsified, a simple signature verification mechanism will not suffice.

5.5. Code Obfuscation

Many apps do not currently use obfuscation and, even if they do, they use the Android-provided Proguard [21]. However, there are many methods of bypassing Proguard, and its security level is very low. Thus, a more secure obfuscation technique should be applied to hide the API name, function name, and variable identifier. As the location of the security mechanism routine can be easily inferred from the log or string variable, it is important to encode the string variable and discard the log. In particular, we could obtain critical information, such as the location and name of the credential, from each app's APIs, which are used to control credential files. Therefore, API hiding should be employed to hide such API invocations and protect against credential cloning attacks.

5.6. Tamper Detection

This method determines whether an application's APK (Android application package) has been tampered with by inspecting the signatures of the primary apps. If used in conjunction with a mobile anti-virus program, this method is predicted to form an effective response. Of course, it is only effective when a forged app is installed.

5.7. Code Attestation

Although obfuscation can cause some difficulties when an attacker uses the protective mechanism of the application layer with static code protection techniques, it is difficult to protect the code when running the app. Supplementing this, an app's code can be dynamically protected during runtime by code attestation, which uses protective hardware such as TEE (Trusted Execution Environment) [22–25] as the trusted point. Using this, not only can the smartphone platform be protected, but, by verifying that an app has been tampered with, a more reliable protective service can be provided.

6. Conclusions

We have examined the weaknesses of the credential management mechanisms in well-known mobile SNS apps using real experimental results. These weaknesses can bring about serious problems such as personal information leakage. When a smart device is lost or stolen, an attacker can use the credential files to gain full access to all of the victim's personal information. Such leaked personal information may be used in cybercrimes such as phishing or spam email. Therefore, to make mobile SNS apps immune to credential cloning attacks, some relevant security mechanisms must be supplemented. We introduced a number of candidate solutions including server-based credential management, session management, code obfuscation, code integrity checking, and code attestation to prevent the simple reverse engineering of Android apps.

Acknowledgments

This research was supported in part by a Global Research Laboratory (GRL) program through the National Research Foundation of Korea (NRF), funded by the Ministry of Science, ICT, and Future Planning (NRF-2014K1A1A2043029), and in part by the Next-Generation Information Computing Development Program through NRF, funded by the Ministry of Science, ICT & Future Planning (2010-0020726).

Author Contributions

All authors have contributed to the study and preparation of the article. All authors have read and approved the final manuscript.

Conflicts of Interest

The authors declare no conflict of interest.

References

1. Kim, H.-I.; Kim, Y.-K.; Chang, J.-W. A grid-based cloaking area creation scheme for continuous lbs queries in distributed systems. *J. Conver.* **2013**, *4*, 23–30.
2. Kim, J.; Yoon, Y.; Yi, K.; Shin, J. Scandal: Static analyzer for detecting privacy leaks in android applications. Presented at the IEEE CS Security and Privacy Workshop (SPW), San Francisco,

- CA, USA, 24–25 May 2012; Available online: <http://www.mostconf.org/2012/papers/26.pdf> (accessed on 8 June 2015).
3. Kumar, K.K.; Geethakumari, G. Detecting misinformation in online social networks using cognitive psychology. *Hum. Centric Comput. Inf. Sci.* **2014**, *4*, 1–22.
 4. Lee, J.D.; Sin, C.H. PPS-RTBF: Privacy protection system for right to be forgotten. *J. Conver. Syst.* **2014**, *5*, 37–40.
 5. Moein, S.; Gebali, F.; Traore, I. Analysis of covert hardware attacks. *J. Conver. Syst.* **2014**, *5*, 26–30.
 6. Abbas, F.; Oh, H. A step towards user privacy while using location-based services. *J. Inf. Process. Syst.* **2014**, *10*, 618–627.
 7. Crowell, A.; Ng, B.H.; Fernandes, E.; Prakash, A. The confinement problem: 40 years later. *J. Inf. Process. Syst.* **2013**, *9*, 189–204.
 8. Enck, W.; Gilbert, P.; Chun, B.-G.; Cox, L.P.; Jung, J.; McDaniel, P.; Sheth, A.N. Taintdroid: An information flow tracking system for real-time privacy monitoring on smartphones. *Commun. ACM* **2014**, *57*, 99–106.
 9. Enck, W.; Ongtang, M.; McDaniel, P.D. Understanding android security. *IEEE Secur. Priv.* **2009**, *7*, 50–57.
 10. Feese, S.; Burscher, M.; Jonas, K.; Troster, G. Sensing spatial and temporal coordination in teams using the smartphone. *Hum. Centric Comput. Inf. Sci.* **2014**, *4*, 1–18.
 11. Jun, K. Push-n-scheme with timeout for content delivery of social networking services. *J. Inf. Process. Syst.* **2014**, *10*, 81–91.
 12. Android Developer. Available online: <http://developer.android.com/training/articles/security-tips.html> (accessed on 5 April 2015).
 13. Jung, J.-H.; Kim, J.Y.; Lee, H.-C.; Yi, J.H. Repackaging attack on android banking applications and its countermeasures. *Wirel. Pers. Commun.* **2013**, *73*, 1421–1437.
 14. Piao, Y.; Jung, J.H.; Yi, J.H. Server-based code obfuscation scheme for apk tamper detection. *Secur. Commun. Netw.* **2014**, doi:10.1002/sec.936.
 15. Bianchi, A.; Oakley, I.; Kostakos, V.; Kwon, D.S. The Phone Lock: Audio and Haptic Shoulder-Surfing Resistant Pin Entry Methods for Mobile Devices. In Proceedings of the 5th International Conference on Tangible, Embedded, and Embodied Interaction, Funchal, Madeira, Portugal, 23–26 January 2011; pp. 197–200.
 16. Kim, S.W.; Yi, H.Y.; Ma, G.I.; Yi, J.H. Shoulder-Surfing Resistant Smartphone Authentication Scheme Using Virtual Joystick, Applied Mechanics and Materials. *Appl. Mech. Mater.* **2013**, *284*, 3497–3501.
 17. Aviv, A.J.; Gibson, K.; Mossop, E.; Blaze, M.; Smith, J.M. Smudge attacks on smartphone touch screens. *WOOT* **2010**, *10*, 1–7. Available online: https://www.usenix.org/legacy/event/woot10/tech/full_papers/Aviv.pdf (accessed on 8 June 2015).
 18. Al Mutawa, N.; Baggili, I.; Marrington, A. Forensic analysis of social networking applications on mobile devices. *Digit. Investig.* **2012**, *9*, S24–S33.
 19. Vidas, T.; Votipka, D.; Christin, N. All Your Droid are belong to us: A Survey of Current Android Attacks. In Proceedings of WOOT’11, the 5th USENIX conference on Offensive technologies, Carrollton, TX, USA, 2011; pp. 81–90.

20. Jeon, J.; Micinski, K.K.; Vaughan, J.A.; Fogel, A.; Reddy, N.; Foster, J.S.; Millstein, T. Dr. Android and Mr. Hide: Fine-Grained Permissions in Android Applications. In Proceedings of the 2nd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices; Raleigh, NC, USA, 16–18 October 2012; pp. 3–14.
21. Lafortune, E. Proguard. Available online: <http://proguard.sourceforge.net> (accessed on 5 April 2015).
22. Loureiro, S.; Molva, R.; Roudier, Y. Mobile code security. In Proceedings of the ISYPAR 2000 (*4ème Ecole d'Informatique des Systèmes Parallèles et Répartis*), Toulouse, France, 1–3 February 2000; p. 46.
23. Jansen, W.A. Countermeasures for mobile agent security. *Comput. Commun.* **2000**, *23*, 1667–1676.
24. Dietrich, K.; Winter, J. Implementation Aspects of Mobile and Embedded Trusted Computing. In *Trusted Computing*, Proceedings of Second International Conference on Trust, Oxford, UK, 6–8 April 2009; Lecture Notes in Computer Science, Volume 5471; Springer: Berlin/Heidelberg, Germany, 2009; pp. 29–44.
25. Sander, T.; Tschudin, C.F. Towards mobile cryptography. In Proceedings of the 1998 IEEE Symposium on Security and Privacy, Oakland, CA, USA, 3–6 May 1998; pp. 215–224.

© 2015 by the authors; licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution license (<http://creativecommons.org/licenses/by/4.0/>).