Hey, You, Get Off of My Clipboard -
On How Usability Trumps Security in Android Password Managers

Sascha Fahl, Marian Harbach, Marten Oltrogge, Thomas Muders, and Matthew Smith

Distributed Computing & Security Group
Leibniz University of Hannover
Hannover, Germany
lastname@dcsec.uni-hannover.de

Abstract. Password managers aim to help users manage their ever increasing number of passwords for online authentication. Since users only have to memorise one master secret to unlock an encrypted password database or key chain storing all their (hopefully) different and strong passwords, password managers are intended to increase username/password security. With mobile Internet usage on the rise, password managers have found their way onto smartphones and tablets. In this paper, we analyse the security of password managers on Android devices. While encryption mechanisms are used to protect credentials, we will show that a usability feature of the investigated mobile password managers puts the users' usernames and passwords at risk. We demonstrate the consequences of our findings by analysing 21 popular free and paid password managers for Android. We then make recommendations how to overcome the current problems and provide an implementation of a secure and usable mobile password manager.

Keywords: Android, Security, Apps, Password Managers, Vulnerability

1 Introduction

Today, using text-based passwords is the most prominent authentication scheme in computer systems. Although researchers have been criticising this scheme as being hard to use in a secure way, it is still the most widely adopted system for authenticating users. Previous research has shown that passwords chosen by users are often easy to compromise by attackers [1,2,3,7,10].

Another problem with the wide-spread application of passwords is password re-use. Users register with many services on the Internet, each requiring them to create a new username/password tuple. Previous research has shown that users often deal with this password overload by re-using the same or similar passwords for multiple accounts [8,13].

Multiple mechanisms have been proposed to support the user in choosing more secure passwords (cf. [12,14]), all trying to alleviate the challenges password-based authentication holds for their users.

However, due to the bounded cognitive abilities and motivation of users, password re-use is still commonplace. Password managers (PMs) aim to overcome this problem: they help the user to handle a large number of different passwords by storing them in encrypted form. To access the encrypted passwords, the user usually has to enter a single master secret that decrypts the password database. Password managers often include a password generator to simplify the creation of new unique and secure passwords. For convenience, login forms are pre-filled and account information for new websites is captured on the fly. Another prominent feature is the synchronisation of password databases between multiple devices.

While early password manager applications were limited to desktop computers and their browsers [11], current implementations offer mobile password manager apps, that can be synchronised over the cloud to ensure that the password database is available on all of the users' devices. While users of desktop password managers benefit from a smooth integration into browsers, password manager apps on mobile platforms offer less comfort. First, web browsers on smartphones and tablets often do not provide a plugin interface, that would allow for a smooth integration of password managers. Second, the existence of dedicated apps for many online services steadily increases the number of users that access online services through an app instead of a website in a general-purpose web browser. This circumstance requires that mobile password managers have to be able to manage passwords not only for browsers but also for apps.

Unfortunately, there is a fundamental problem with password manager apps on Android: The OS does not offer an API to integrate password managers with the browser or other apps. This has led to the adoption of a highly insecure practice to overcome this weakness: Password managers use the OS clipboard to transfer credentials from a password manager app to the browser or other apps. This method effectively broadcasts credentials to all apps installed on the smartphone.

We analysed the security of 21 password manager apps on Android, having a combined install base of 2,500,000 to 10,900,000 devices[1]. We found that apps mostly use AES, Blowfish or a combination of both to encrypt credential databases, although some apps use their own crypto implementations and do not rely on a proven open-source library. Seven apps use a key derivation function to derive the symmetric encryption key from the user's master secret to strengthen the security of encrypted credentials. We found one PM that directly inputs the user's password as the encryption key and truncates passwords longer than 16 characters. In case the password has less than 16 characters, the string '`FEDCBA9876543210`" is appended to "*strengthen*" the password. Another PM app uses an HMAC algorithm with SHA-256 and the fixed initialisation vector `''notverysecretiv''` for key derivation for every encrypted credential tuple. Three password manager apps provide their own cloud synchronisation feature to easily share passwords between multiple devices. Two of them synchronise the users' databases over a broken TLS channel (i. e. the apps are vulnerable

[1] The numbers are based on information provided by Google's Play Market. Google does not provide more fine-grained numbers for an app's install base.

due to incomplete TLS certificate validation) and hence allow a Man-In-The-Middle attacker to capture the databases. We investigated this problem in [5] in more detail and analysed 13,500 popular Android apps and found that many app developers failed to apply TLS appropriately. However, most critically, all apps use the clipboard feature to transfer credentials from the password manager to browsers or other apps. Two apps automatically copy credentials to the clipboard when the user clicks the URL for the given online account. Only one app allows a user to disable the clipboard feature.

While there is known malware on desktop computers that threatens passwords copied to the computer's clipboard[2], this circumstance has not been investigated in detail yet. Hence, to the best knowledge of the authors, this paper is the first analysis of password managers' security on mobile devices.

The remainder of the paper is organised as follows: Section 2 gives background information on password managers and discusses peculiarities of password managers on desktops and mobile devices. Section 3 describes our attack against password managers on Android and illustrates how captured credentials can be linked to online services and how stolen information can be transferred to the attacker's server without giving the user a chance to notice. Section 4 gives more details on the security of the PMs we analysed. To understand why developers added the clipboard features to their apps, Section 5 summarises open interviews with app developers. Section 6 discusses countermeasures and presents an implementation of a secure and usable password manager that overcomes the presented vulnerabilities. In Section 7, we conclude the paper and discuss future work.

2 Background

While early password managers were simply a username/password database embedded in desktop computers' browsers, the features of modern password managers are much more extensive and can be categorised as follows:

Browser-Embedded PM: Most web browsers, such as Google Chrome, Microsoft Internet Explorer, Mozilla Firefox or Apple's Safari, include an embedded password management feature. In case a user logs into a website for the first time, the PM inquires whether the login credentials should be saved to ease future logins by automatically filling the username and password into the login form. These embedded password managers traditionally store credentials locally on the user's computer, but are increasingly syncing them between multiple devices using proprietary Cloud services. Some browsers do not encrypt credentials stored locally or require the user to set a master secret to enable encryption. For example, Chrome uses the Google Account password to encrypt the synced password database by default, but offers to use a dedicated secret as an advanced feature.

[2] e. g.: http://www.f-secure.com/v-descs/trojan_w32_qhost_je.shtml

Browser Plugins The majority of modern desktop web browsers provide an API to extend their functionality by allowing the user to install third-party plugins or extensions. Many password managers are hence available as browser plugins. KeePass[3], 1Password[4] and Lastpass[5] are prominent examples of plugin-based password managers. They encrypt passwords and protect them with a master secret. These third-party password managers often provide further functionality and act as encrypted storage for more than just usernames and passwords: other sensitive information such as credit card numbers, online banking information or secret notes can be stored as well.

2.1 Password Managers on Desktops

Password managers on desktop computers are generally well integrated into the users' everyday Internet-facing software, such as browsers and email clients. Regardless of whether an embedded password manager or a third-party plugin is used, when the user accesses an online account for the first time or creates a new account, the password manager automatically comes into play and offers the user to securely store the new account information. The plugin APIs of modern browsers offer a very comfortable integration of password managers into the users' workflows. When a user visits a website that requires authentication, a password manager typically auto-fills the username and password and might even automatically submit the login form.

Early password managers for desktop computers (e. g. [11]) assumed a single device environment. Nowadays users often work with multiple devices such as desktops, notebooks, smartphones and tablet PCs, which makes it necessary to synchronise password databases between multiple devices to have credentials available whenever needed. For this reason, some password managers offer to sync databases between multiple devices by storing credentials in the Cloud or by putting the database on USB drives. A popular way to sync password databases is the Dropbox service. The encrypted password database is stored in the user's Dropbox account and can be accessed from all the user's devices. 1Password for example maintains an encrypted database for sensitive information and allows users to store and share the database via their Dropbox account.

2.2 Password Managers on Mobile Devices

While password managers in desktop environments are well integrated into browsers and users' workflows, the situation for third-party password managers on mobile platforms is different. Neither of the major mobile platforms (Android, iOS and Windows Mobile) nor mobile browsers provide a plugin API comparable to desktop computers. Additionally, the paradigm shift away from the browser

[3] cf.: http://keepass.info/
[4] cf.: https://agilebits.com/onepassword
[5] cf.: https://lastpass.com

as a generic tool to surf the Internet towards the „there is an app for everything" approach makes integrating PMs into mobile ecosystems even harder.

API limitations and the requirement to support arbitrary apps creates a different usage pattern for mobile PMs. Instead of storing new account information automatically and auto-filling authentication forms, the workflows of mobile PMs typically consist of the following steps:

1. The user has to switch to the PM app,
2. then needs to find the appropriate username/password tuple from a list of stored credentials,
3. copies the password to the clipboard,
4. switches back to the app that requires authentication, and
5. finally pastes the password into the corresponding text field before submitting the login form.
6. In case the user does not remember the username for a given service, these steps (except step 2) are repeated for the username as well.

Although this workflow's usability is far from optimal, it is the best mobile password managers can provide so far. To understand why users nevertheless use password managers on mobile devices, we analysed 2,000 user reviews in Google's Play Market. To this end, we downloaded user reviews, manually extracted factors that motivate users to use PMs on their Android device and identified the following reasons to be substantial:

Protection: Users feel that embedded PMs do not store the passwords in a way they believe to be secure (e. g. some users were angry that Android's stock browser does not encrypt stored usernames and passwords).

Confidentiality Users do not trust embedded PMs in keeping their data confidential (e. g. users were afraid that their credentials could be sent to Google).

Features: Embedded PMs are usually limited to usernames/passwords. Users often want to store other confidential data, such as banking information.

Availability: Embedded PMs are usually limited to a single browser. Since many users need access to their information on multiple devices and browsers, a vendor-independent PM is preferred.

After having outlined background information on password managers, the next section presents our attack on PM apps on Android and illustrates its consequences for the user.

3 Password Sniffing on Android

As illustrated in Section 2.2, the workflow of mobile password managers requires the user to copy account credentials to the clipboard before switching to the target app and pasting them before actually logging in. There are some problems with this practice: On Android, writing data to or reading data from the clipboard does not require any permission. Therefore, every app currently running on an Android device can read the items stored in the clipboard at any time.

To make matters worse for password managers, the Android SDK provides the `android.content.ClipboardManager.OnPrimaryClipChangedListener` interface, which defines a listener callback that is invoked each time the primary item on the clipboard changes. This can be used by malicious apps to harvest passwords as they are passed through the clipboard. As a proof of concept we implemented a password sniffer named *PWSniff* using this mechanism. PWSniff runs as a background service and does not require any Android permission to work properly.

As long as no changes to the clipboard occur, the background service idles and therefore does not consume any CPU cycles. Directly after a new item is copied to the clipboard, the listener callback is invoked by Android and the idling PWSniff background service is notified and then reads the primary item. Next, PWSniff determines the app which is currently in the foreground. This information can also be acquired without requesting any permission. We assume that the foreground app at the time of copying is the app from which a user copied data (cf. Section 2.2 step 1). In case this app is a known password manager, we assume that the primary clipboard item is either a service URL, a username or a password (cf. Section 2.2 step 3). Whether the app is a password manager can be determined based on the app's user ID, which is assigned at install time and can be mapped to the a unique app market ID. The third step in our attack is to wait for a foreground app switch by checking the current foreground app in a loop and waiting until the user brings another app to the foreground. In case we identified the primary clipboard item as possibly confidential data (no matter if it is a username or a password) copied from a password manager, the new foreground app is assumed to be the destination of the credentials-copy-operation (cf. Section 2.2 step 4). Hence, by exploiting features of the Android SDK that require no special permissions in combination with a typical workflow in the context of using password manager apps on Android, it is easily possible to harvest (still potentially noisy) usernames and passwords from the world readable clipboard.

At this point, an attacker cannot be sure which item is the username and which the password. But, in many cases it is possible to differentiate between both items based on their structure. Usernames are often chosen to be easily memorable (e. g. an email address) while passwords, especially those which are managed with PM software, usually are more "cryptic". Even in cases where the username and password cannot be easily distinguished, an attacker could first try one combination of the sniffed items and in a second attempt the reversed order. In both cases, breaking into an account is straightforward.

Advanced Username Capture: The attack described above relies on the user copying and pasting both the username and the password. Since users might just type their usernames from memory or use browser or app autofill features to save this effort, it might become necessary to acquire the username through an alternative method. For this, PWSniff can be equipped with the GET_ACCOUNTS permission. The permission allows the app to see usernames that other apps handle

on the smartphone and which are registered with the *AccountManager*[6]. This also includes all email addresses used on the device. Since many online services use email addresses as usernames, this list offers a good basis from which to guess usernames for many services.

The downside of this extension is that it involves the danger of a user becoming suspicious of the app's permissions, which are presented to the user at install time. However, Felt et al. [6] demonstrated that users pay little attention to the permissions of an app and mostly do not understand the permissions' meaning. While Felt et al's. results account for Android's permission system in general, an app's permissions are also grouped and classified based on their security relevance. In this respect, the GET_ACCOUNTS permission does not rank particularly high and thus is not often shown on the first page. To ensure that the GET_ACCOUNTS permission is not shown on the first page, an attacker only needs to add more than three popular permissions such as INTERNET, LOCATION and STORAGE which are used by many apps to his malware app. The best composition of permissions to mask the GET_ACCOUNTS permission is outside the scope of this work.

Advanced Account Capture: In Section 3, we illustrated that an Android app which holds no special permissions is able to sniff online account credentials that are copied to the clipboard when working with any password manager on Android in most cases. It is also possible to learn from which app a value was copied to the clipboard and into which app the value was pasted. If the target app has a special purpose (e. g. the Skype app only logs into Skype), it is easy to guess to which online service the harvested credentials belong. However, in case the target app is a multi-purpose Internet client such as a web browser, finding the intended service is not quite as straightforward.

To learn for which account a password is used, an attacker can benefit from Android's ProcFS features. The ProcFS is an interface to the kernel and provides information about a device such as information about the CPU, memory and network details. On Linux-based systems such as Android, the ProcFS is usually mounted at `/proc`. Most entries in `/proc` and its subdirectories can be read by everyone. The `/proc/net/tcp` file contains information about all TCP connections on an Android device and is also world-readable and hence accessible by every app without requiring any permissions. Information such as source IP and port, destination IP and port and the UID of the process that created the network connection are listed there. Since Android creates a static mapping of Apps to a UID at install time, one can easily learn which app connects to which Internet hosts based on the UID entry in `/proc/net/tcp`. Having the destination IP for an app's network connection at hand allows an attacker to easily infer to which online service a credential pair is connected by logging all network connections of an app, immediately after a copy operation from a PM to another app was discovered.

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

Exfiltrating the Data: In [5], we found that 92.8 % of 13,500 popular Android apps request Internet access. Adding the Internet permission to PWSniff should thus not raise undue concern. With this permission, transmitting the harvested data is of course trivial. However, if a zero permission attack app is desired, exfiltration of the harvested data can still be done using another flaw in Android's permission system. Egners et al. [4] describe a loophole in Android's permission system that we adopt for our purposes and which allows PWSniff to send gathered credential information to a remote HTTP server without requiring the Internet permission. After the account login information was gathered, the harvested data is cached until the device's display is turned off. When this happens, an HTTP URL with the following structure is built: `http://<pwsniff-master>/pw#username#service`. This URL is used to invisibly open Android's stock browser when the display is turned off by running the following code in PWSniff. We explicitly call Android's stock browser since some third-party browsers do not hand back control for unknown protocols to the Android OS, which is required to keep the attack stealthy.

The server behind the URL replies with a location header containing a custom protocol, for example: `'Location:pwsniff://all.ok'`. Since PWSniff includes an activity that previously registered for the custom `pwsniff://` protocol, the browser passes handling for the URI `pwsniff://all.ok` to PWSniff. Staying invisible, the activity then simply terminates.

After demonstrating how credentials can be sniffed when Android password managers are used, how they can be mapped to online accounts and how this information can be exfiltrated stealthily, the next section gives some relevant excerpts of our detailed security analysis of PMs on Android.

4 Security in Detail

We analysed 13 free and 8 paid Android PM apps in detail. Our intention was to analyse which apps include the clipboard feature for credential copy & paste, which encryption algorithms protect the password database, whether or not the app includes an embedded browser, whether or not the SD card is used to store the password database and whether or not the app removes itself from the recent apps view. For analysis, we installed all apps on a Samsung Galaxy Nexus with Android 4.0. We applied forensic techniques[7] to learn database and configuration files' structures of the installed password manager apps. To learn internals of the password managers, we decompiled them[8] and conducted manual static code analysis.

[7] We used the adb tool (cf.: http://developer.android.com/tools/help/adb.html) for logical extraction.

[8] We used a bundle of decompilation tools: JD-GUI (cf.: http://java.decompiler.free.fr/?q=jdgui), apktool (cf.: http://code.google.com/p/android-apktool/) and dex2jar (cf.: http://code.google.com/p/android-apktool/)

We also conducted static code analysis on the same dataset as in [5] and found that only two apps in this dataset registered for the clipboard change listener. We analysed both apps manually and found no malicious behaviour in the apps. 907 apps (6.7 %) in the sample access the clipboard API programatically to share more complex objects than simple text strings such as images, video or audiofiles.

Table 1 in the Appendix shows an overview of the security parameters we analysed.

4.1 Encryption

One important aspect of PM security is the encryption mechanism to store credential databases. Android's stock browser does not encrypt stored passwords in any way but protects them from unauthorised access by file system permissions. Android's AccountManager mechanism provides centralised credential storage and also protects user credentials from unauthorised access by file system permissions, but the `accounts.db` database is not protected with an extra layer of encryption. This does not protect the password from forensic analysis.

All third party PMs we analysed apply some encryption mechanism to protect the data. Android supports (3)DES, RC2, and RC5[9] to encrypt data out of the box. Other encryption algorithms require the developer to add third-party libraries to their app. We decompiled the PMs to find out what kind of encryption algorithm is applied in each PM app. To provide stronger security, most password managing apps use the Advanced Encryption Standard (AES) with several key lengths. aWallet uses a combination of AES, Blowfish and 3DES.

A critical aspect of encrypting password databases is the derivation of the encryption key [9] that is directly connected to the master secret used to unlock/decrypt the password database. Seven apps use a dedicated key derivation function to derive the symmetric encryption key from the user's master secret to strengthen the security of encrypted credentials. We found one app that directly inputs the user's password as the encryption key, truncating passwords longer than 16 characters. In case the password has less than 16 characters, the string "FEDCBA9876543210" is appended to *"strengthen"* the password. Another app uses an HMAC algorithm with SHA-256 and the fixed initialization vector `''notverysecretiv''` for key derivation.

4.2 Storage

Most password managers, including Android's stock browser, store password databases in files or SQLite databases that are only accessible by the password manager app itself. Hence, other Android apps cannot access account information regardless of whether it is encrypted or not. Ten of the analysed PMs

[9] cf.: http://developer.android.com/reference/javax/crypto/spec/package-summary.html

store databases on the SD card that is world readable without requiring further permissions on all devices with Android 4.0 and older. In combination with inappropriate database structures (not encrypting all information stored in the password manager), an attacker is for instance able to learn for which services a user holds accounts or for which services the same username and/or passwords are used.

4.3 Recent Apps

An essential feature of Android devices is an overview of the currently running apps, also called the *Recent Apps View*. The Recent Apps View shows thumbnails of current foreground activities of all running apps. While a security feature of all analysed password managers is the automatic locking of the password database either immediately after the password manager app was left or after a configurable amount of time, we found only three apps that also replace their thumbnails in the recent apps view (cf. Table 1). In case the user copied online account information (usually the location, username and password) and then leaves the password manager app to paste the information into another app, the account information is left in the recent apps view and can be seen by anyone with physical access to the user's device. Although this threat is orthogonal to our attack (cf. Section 3), it outlines a security risk for users' online credentials.

4.4 Cloud Sync

While all password managers store their databases locally and allow synchronisation of their databases, most offer a more manual functionality using Dropbox or similar services. LastPass, SecureSafe and RoboForm provide dedicated Cloud storage features to automatically synchronise all passwords remotely. In [5] we analysed popular Android apps and found that many app developers fail to apply TLS appropriately, being vulnerable to active Man-In-The-Middle attacks. Although LastPass, SecureSafe and RoboForm protect their network communication with TLS, SecureSafe and RoboForm fail to verify the cloud servers' TLS certificates. Instead, they accept all certificates. In case of SecureSafe this however has not further security implications since in addition to TLS, SecureSafe uses a session-specific symmetric key, which is set up during the SRP-login[10], to additionally encrypt password-data end-to-end. However, RoboForm leaks the users' credentials which are used for password encryption in the default case (i. e. the user did not choose an extra password for encryption). Hence, an attacker can gain access to the data in cleartext under this circumstances.

5 The Developers View

After analysing Android password managers on a technical basis, we contacted their developers via email and informed them about a possible security threat

[10] cf. RFC2945

for their users. We offered them to get in contact either via email or telephone to discuss the details of the PWSniff attack. We also posed the following questions:

- Why was the C&P feature used in the password manager app?
- Were developers aware of the security threats arising from using the clipboard for username/password sharing, and, if so, why did they add the C&P feature nonetheless?
- Which features, if any, do developers miss in Android's SDK for developing a password manager app?

15 of the 21 developers agreed to participate in the email interview and are anonymously referred to as P1, ..., P15 in the following.

5.1 Results

During the discussions with developers, we were able to identify three different reasons to add the usability-enhancing clipboard feature to PM apps. One was because the developers themselves were users of their apps and desired the feature themselves. ("As I'm a [...] program user too, I added the copy feature because I needed to transfer usernames (that are usually long email addresses) and passwords to login forms in web browsers."; P7). The second reason provided by PM developers was the wish to come as close as possible to PM functionality on the desktop, because developers believed that users would reject their apps if they were not sufficiently usable. ("Copy to clipboard has been in [...] Android from early on. [...] It was something that we knew we needed to make the application usable at all"; P4). Lastly, developers reported that users directly requested a C&P feature for their app ("The feature was highly requested by users. The most common example: users want to login to a website on their mobile device, so he/she copies credentials from [our PM] to the clipboard and then pastes them into the browser."; P15).

All but one developer were aware of security threats resulting from putting passwords into a device's clipboard. Developers who were aware of the security threat justified adding the clipboard integration, stating that they had no other choice. They described it as a tradeoff between usability and security which was decided in favour of increasing usability ("It's a balance between ease of use and security. Of course it would be much more secure to not use the clipboard, however people accept the risk of doing so; the alternative of not using a password manager is worse."; P3). One developer interestingly described his decision not as a usability-security tradeoff but as a "one type of security versus another type of security" decision, alluding to the fact that without password managers users would choose less secure passwords. Additionally, P4 stated: "On the whole, I think that password reuse [...] is currently the biggest single problem with password security today. And so, if a password manager gets people to use unique passwords for each site, the dangers of a publicly readable clipboard is a security risk that can be worthwhile. [...] What's the alternative?".

All developers criticised Android's missing support for password manager apps. A native integration into third party apps and browsers was described as

the most effective countermeasure against the password sniffing security threat ("Android doesn't offer hooks into the native default browser [...] and does not allow our app to access input fields of other apps [...] which makes it necessary that password managers make heavy use of the clipboard."; P3).

5.2 Discussion

Based on the lack of API support for third-party password managers on the Android OS, developers decided to opt for the best usability they could achieve by including the clipboard feature to allow users to copy-and-paste usernames and passwords from their apps to other apps. Although all but one developer were aware of the possible security threat, they decided that better usability was more important than stronger security. A justification multiple developers offered was that *they had no other choice* and that it was necessary to add the best possible usability even if security was threatened.

6 Countermeasures

With the results of our analysis and the developers' comments in mind, we first discuss possible countermeasures to improve the security of a smartphone's clipboard facilities as a global shared memory. Additionally, we present a PM implementation for Android based on a customised soft-keyboard that provides usability features similar to desktop PMs and does not leak credentials over public channels.

Secure Clipboard Architecture: Sniffing confidential information on Android devices is currently easy since on the one hand, a proper plugin API for integrating password managers is missing and, on the other hand, the design of the current clipboard mechanism on Android is not made for sharing confidential information between apps. The current clipboard model allows an arbitrary app to access clipboard items deposited by any other app. With the assumption that both, the copy as well as the paste operation are triggered by the user, such a clipboard model does not cause security concerns. However, on Android, two other API features open the door for malicious activity: Android's background service feature for apps and the `ClipboardManager.OnPrimaryClipChangedListener` allow for stealthy harvesting of clipboard items (cf. Section 3). Therefore, we present two possible modifications to improve Android's clipboard model when it is accessed using API functionalities:

Permissions The current clipboard model allows every app to programmatically read data from and write data to the clipboard, without requiring permission for that. While user-triggered clipboard operations can remain unchanged, we propose two new permissions for API-based access to clipboard functionality: `WRITE_TO_CLIPBOARD` and `READ_FROM_CLIPBOARD`. Although the limited effects of Android's permission model for the average

app user have been discussed (cf. Felt et al. [6]), these permissions should be added for completeness. This way, at least the tech-savvy users would have a chance to see if an app is capable of accessing the clipboard programmatically and can warn the rest of the community. Since we identified only very few apps to access the clipboard programatically (cf. Section 4), the proposed changes would only impact a small number of apps. Regular, user-triggered copy-and-paste operations would not be influenced by this modification.

Targeted Clipboard Copying a value to the clipboard on current Android smartphones is equivalent to broadcasting the information to all other apps. This is contrary to the users' intuition of using a copy-and-paste feature that is generally used to transfer information from one app to another. Therefore we propose to extend API calls to the clipboard with a "target app" parameter that the app may request from the user. Keeping usability in mind, the number of target apps should be kept to a minimum. Apps providing an API-based copy feature may let the user choose target apps from a list of all apps or suggest useful targets as well as remember previous preferences. If clipboard operations are triggered by the user, reading the clipboard's contents should only be possible through explicit user interaction as well.

The modifications to Android's current clipboard model proposed above do not only protect credentials from unwanted disclosure, but can also serve to shield any other (possibly confidential) information (such as financial or medical information), that a user might copy to the clipboard.

USecPassBoard: While the above solutions would alleviate the current security problems of PMs, they would also require modification of the Android OS itself. Additionally, these measures cannot address the usability issues of mobile PMs, i. e. that the user needs to manually select credentials, switch apps and manually paste. To offer both better security *and* usability we propose a novel password manager: USecPassBoard. To overcome the issues plaguing the traditional approach of mobile password managers, we went down a different path. We created a soft-keyboard which integrates a password manager. Since soft-keyboards are available in every app and can access a shared credential database, they integrate well with most scenarios where credentials need to be entered. A custom soft-keyboard implementation on Android replaces the default keyboard and provides a custom means to input data into user-input fields. Figure 1 in the Appendix shows the user interface of the USecPassBoard PM. Besides preserving the regular keyboard functionality, it essentially adds two operations: (1) Creating a new username/password entry and (2) inserting a username/password tuple at the user's discretion. Since USecPassBoard is a soft-keyboard, it is available in every application, including the browser and stores passwords in a master secret-protected AES-256 encrypted database[11] to protect username/password tuples from unauthorized access. This effectively avoids the use of copy-and-paste on usernames and passwords while maintaining the flexibility of all available password managers.

[11] We use the SQLCipher (cf.: http://sqlcipher.net/sqlcipher-for-android/) database.

New Account: USecPassBoard analyses the context of user input to determine if credentials are being entered. The password context is determined by identifying which app is currently used (i. e. which app is in the foreground) and in case the foreground app is a browser, it determines the website which is displayed by reading the browser's first item cached in the history. Apps are uniquely identified based on their package names managed by the Android operating system[12]. USecPassBoard then caches the input of all textfields in the foreground activity. This is possible since soft-keyboards on Android are triggered when a textfield is activated by the user. Additionally, a soft-keyboard receives a reference of the `EditorInfo` class[13] which identifies an input as a text or a password field. After the user completes the input and the keyboard loses the focus on a password field, a notification is displayed in the status bar that a new dataset was created (cf. Figure 1) if there is no identical username/password tuple for the current context in the database. New username/password tuples are bound to the target app – based on its package name – and are not available for possibly malicious apps. In case the user would like to share credentials between different apps (e. g. between two Facebook client apps), we allow this in the settings menu of USecPassBoard.

Credential Insertion: In case USecPassBoard recognises a known password context (i. e. a package name of an app for which credentials are stored in the database), the user can choose to insert this information by tapping into the input field for the username or password. A popup message appears after the user tapped onto the key button (cf. Figure 1) and a list of available credentials for the given password context is displayed. Subsequently the selected username/-password tuple is inserted at the user's discretion and the login process can be started.

Security Considerations: All interactions between the USecPassBoard virtual keyboard and a target app must be initiated by the user by tapping into a text input field. This creates a communication channel between the keyboard and the target app through Android's InputMethodManager[14] which is not accessible from other third party apps. This allows the automatic storage of new account credentials and insertion of stored credentials into uniquely identifiable target apps.

Target apps are uniquely identified based on their package name that is managed by the Android OS and cannot be spoofed by malicious apps[15]. In case the target app is the browser, a password context consists of the browser's package name and a target website. We identify the target website by reading the top item from the browser's history. This is accessible with Android's READ_HISTORY_BOOKMARKS permission and gives us the currently viewed website. Hence, we can avoid that users falsely insert credentials another website.

[12] cf.: http://developer.android.com/guide/topics/manifest/manifest-element.html
[13] cf.: https://developer.android.com/reference/android/view/inputmethod/EditorInfo.html
[14] cf.: http://developer.android.com/reference/android/view/inputmethod/InputMethodManager.html
[15] cf.: http://source.android.com/tech/security/

7 Conclusions

With the rise of mobile devices, mobile password manager apps could be an integral security tool for smartphone and tablet PC users. Since Android based devices lack APIs for the integration of password managers, current solutions rely heavily on the clipboard to share credentials between the PM and other apps. We analysed 21 popular password managers on Android which all are vulnerable to credential sniffing because a device's clipboard is a publicly available storage that can be accessed from any app. We showed that, using additional context information, malware is able to link the stolen credentials to the corresponding online account in many cases. We interviewed developers of the analysed PM apps and found that the majority of them were aware of possible security threats but accepted the risk to provide better usability. Based on the analyses' findings and developers' feedback, we discuss modifications to Android's clipboard mechanism to increase security for sensitive information. Finally, we present a soft-keyboard that integrates a secure and easy-to-use password manager which prevents the leakage of usernames/passwords via the clipboard. This password manager design is the first to offer both usability and security for Android-based password managers.

Since, in addition to security, usability is crucial for a password manager, in future work we plan to conduct multiple user studies for the USecPassBoard password manager.

References

1. M. Bishop and D. V Klein. Improving system security via proactive password checking. *Computers & Security*, 14(3):233–249, 1995.
2. J. Bonneau. The Science of Guessing: Analyzing an Anonymized Corpus of 70 Million Passwords. *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 538–552, 2012.
3. M. Dell'Amico, P. Michiardi, and Y. Roudier. Password Strength: An Empirical Analysis. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9, 2010.
4. A. Egners, B. Marschollek, and U. Meyer. Messing with Android's Permission Model. In *IEEE International Conference on Trust, Security and Privacy in Computing and Communications, (IEEE TrustCom-12)*, May 2012.
5. S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why eve and mallory love android: an analysis of android ssl (in)security. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 50–61, New York, NY, USA, 2012. ACM.
6. A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: user attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, pages 3:1–3:14, New York, NY, USA, 2012. ACM.
7. D. Florencio and C. Herley. A large-scale study of web password habits. *Proceedings of the 16th international conference on World Wide Web*, pages 657–666, 2007.
8. S. Gaw and E. W. Felten. Password management strategies for online accounts. In *Proceedings of the second symposium on Usable privacy and security*, SOUPS '06, pages 44–55, New York, NY, USA, 2006. ACM.
9. B. Kaliski. PKCS #5: Password-Based cryptography specification version 2.0. RFC 2898, Internet Engineering Task Force, Sept. 2000.
10. D. Malone and K. Maher. Investigating the distribution of password choices. In *Proceedings of the 21st international conference on World Wide Web*, WWW '12, pages 301–310, New York, NY, USA, 2012. ACM.
11. B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. C. Mitchell. Stronger password authentication using browser extensions. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association.
12. R. Shay, P. G. Kelley, S. Komanduri, M. L. Mazurek, B. Ur, T. Vidas, L. Bauer, N. Christin, and L. F. Cranor. Correct horse battery staple: exploring the usability of system-assigned passphrases. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, pages 7:1–7:20, New York, NY, USA, 2012. ACM.
13. R. Shay, S. Komanduri, P. G. Kelley, P. G. Leon, M. L. Mazurek, L. Bauer, N. Christin, and L. F. Cranor. Encountering stronger password requirements: user attitudes and behaviors. In *Proceedings of the Sixth Symposium on Usable Privacy and Security*, SOUPS '10, pages 2:1–2:20, New York, NY, USA, 2010. ACM.
14. B. Ur, P. G. Kelley, S. Komanduri, J. Lee, M. Maass, M. L. Mazurek, T. Passaro, R. Shay, T. Vidas, L. Bauer, N. Christin, and L. F. Cranor. How does your password measure up? the effect of strength meters on password creation. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 5–5, Berkeley, CA, USA, 2012. USENIX Association.

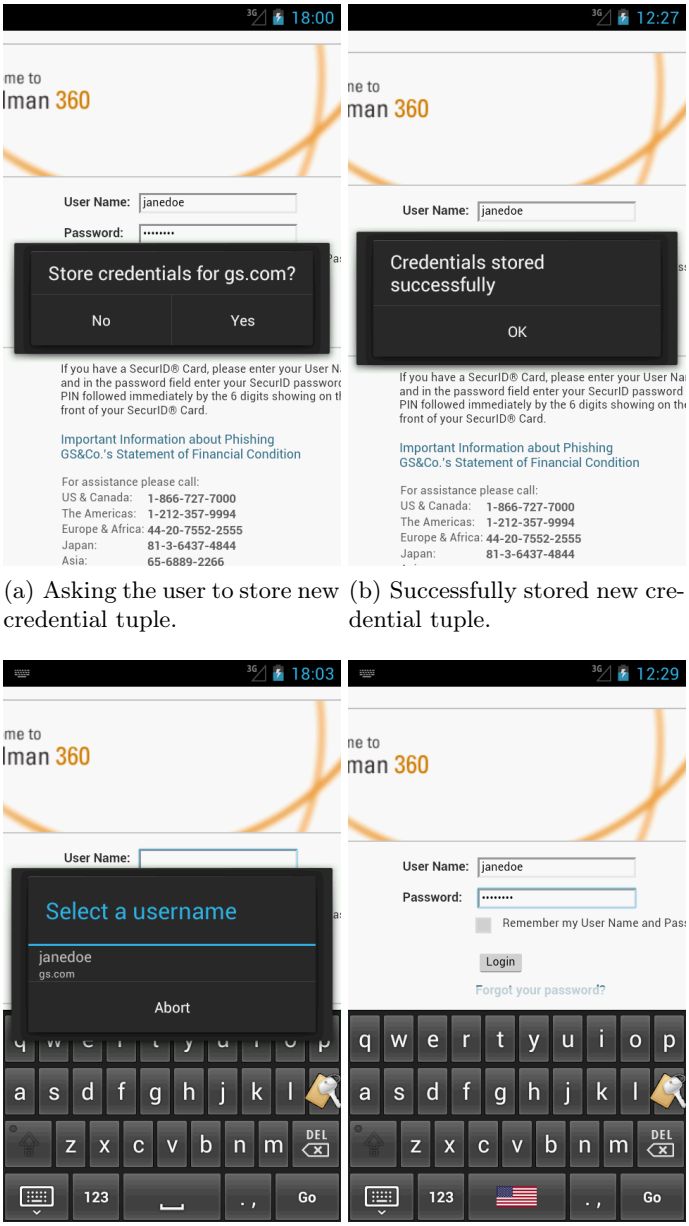
A Investigated Password Managers

Table 1. Overview of the analysed Android password manager apps. (EM=Encryption Method, KD=Key Derivation, C&P=copy&paste functionality, EB=Embedded Browser, SD=Writes database to SD card, RA=Removes itself from the recent apps view)

Free							
App	**Installs**[1]	**EM**	**KD**	**C&P**	**EB**	**SD**	**RA**
PassDroid	100-500k	AES	SHA-256	✓	–	Backup	✓
1Password	100-500k	AES	PBKDF2	✓	✓	Always	✓
KeePassDroid	500k-1m	AES[1]	SHA-256	✓	–	Always	✓
UPM	100-500k	AES	PBE	✓	–	Backup	✓
Pocket	100-500k	AES	PBE	✓	–	Backup	✓
NS Wallet	10-50k	AES	–	✓	–	✓	✓
LastPass	100-500k	AES	○[4]	✓	✓	–	–
PasswdSafe	10-50k	AES	–	✓	–	✓	✓
OI Safe	100-500k	AES	PBE	✓	–	✓	✓
aWallet	100-500k	AES[2]	SHA-256	✓	–	Backup	✓
Moxier Wallet	10-50k	AES	SHA-256	✓	–	–	✓
Keeper	1-5m	AES	SHA-1	✓	✓	Backup	✓
RoboForm	100-500k	○[4]	○[4]	✓	✓	Backup	✓
Paid							
mSecure	100-500k	Blowfish	SHA-256	✓	–	✓	–
Secret Safe	50-100k	AES[3]	SHA-256[3]	✓	–	Backup	✓
SafeWallet	10-50k	AES	HmacSHA1	✓	–	✓	✓
SPB Wallet	10-50k	AES	○[4]	✓	–	✓	✓
eWallet	10-50k	AES	PBE	✓	–	✓	–
Handy Safe Pro	10-50k	Blowfish	–	✓	–	–	✓
DataVault	1-5k	AES	–	✓	–	Backup	✓
Password Box	5-10k	AES	–	✓	–	✓	✓

[1] KeePassDroid combines AES and Twofish
[2] aWallet combines AES, Blowfish and 3DES
[3] Secret Safe combines AES and Twofish for encryption and multiple rounds of SHA-256 and Whirlpool for key derivation.
[4] This information could not be found by reverse engineering.

B USecPassBoard User Interface



(a) Asking the user to store new credential tuple.

(b) Successfully stored new credential tuple.

(c) Selecting existing credential tuple.

(d) Credentials filled in.

Fig. 1. The USecPassBoard workflow for storing new credential tuples and filling in stored credentials.