A keyboard that manages your passwords in Android

Faysal Boukayoua
KU Leuven
Dept. of Computer Science
Technology Campus Ghent
Gebroeders De Smetstraat 1
9000 Ghent, Belgium
faysal.boukayoua **at** cs.kuleuven.be

Bart De Decker
KU Leuven
Dept. of Computer Science iMinds-DistriNet
Celestijnenlaan 200A
3001 Heverlee, Belgium
bart.dedecker **at** cs.kuleuven.be

Vincent Naessens
KU Leuven
Dept. of Computer Science
Technology Campus Ghent
Gebroeders De Smetstraat 1
9000 Ghent, Belgium
vincent.naessens **at** cs.kuleuven.be

***Abstract*—During the recent years, smartphones and tablets have become a fixture of daily life. They are used to run ever more tasks and services. Unfortunately, when it comes to password management, users are confronted with greater security and usability concerns than in the non-mobile world. This work presents a password manager for Android that can accommodate any app. Existing platform mechanisms are leveraged to better protect against malware and device theft, than current solutions. Our approach also provides significant usability improvements. No modifications are required to existing applications or to the mobile platform.**

I. Introduction

In the recent years mobile phones and tablets have become commonplaces of everyday life. They are increasingly used to run apps and Web services. Similarly to desktops and laptops, security remains important. The stakes are even higher, as mobile devices are more prone to theft and loss. Furthermore, users are still required to use passwords, since Web services and apps continue to rely on them [6].

There is a limit on the strength and the number of passwords that humans can retain [8], [9]. Hence, many users choose weak passwords and reuse existing ones. In addition, many mobile password managers on the market offer questionable security [2], [14].

Despite innovations in human-machine interaction, password entry remains cumbersome [10]. Different keyboard layouts and modes of operation per key, slow down typing. Also, people with thick fingers experience precision issues when using touchscreens [12]. These usability shortcomings adversely affect password best practices.

Contribution This paper proposes a mobile password management keyboard that accommodates any Android application. Storage, management and provisioning are centralised in one dedicated app. The approach taps into existing platform facilities to provide better protection against malware and device theft, than existing solutions. Compared to widely-used mobile vault apps, password storage and provisioning is more secure. The user's workflow is minimally disrupted, as the functionality is directly available from the keyboard. Unlike browser-based solutions, our approach is not limited to the passwords of a single application. Ultimately, the barrier is lowered towards working with stronger, more differentiated passwords. Neither the platform nor the relying apps require modification. A prototype validates our approach.

This paper is structured as follows. Section II presents related work, followed by a description of the approach in section III. Subsequently, section IV elaborates on the prototype. An evaluation is made and extensions are suggested in section V. Finally, conclusions are drawn in section VI.

II. Related work

Password-based authentication incurs various usability and security issues [6], [8], [9]. These are further exacerbated by the nature of mobile keyboards.

A first category of solutions addresses the problem by facilitating authentication at the server-side. Single sign-on is a well-established concept, that allows users to access multiple services with one authentication. Examples are Kerberos, Shibboleth [13], OpenID [7] and proprietary APIs like Google's and Facebook's. Other credential types can be used too, f.i. certificates and one-time password generators. Lastly, a concept that improves usability for touchscreens, are the image-based authentication schemes [16]. However, server-side mechanisms are no systematic strategy for users: they require the cooperation of authentication providers.

A second class of approaches is applied to the mobile itself. LastPass and SafeWallet are examples of browser-based password managers. They offer the convenience of portability and effortless backup across instances of the same browser plugin. However, passwords outside the browser are disregarded. This shortcoming is significant, since countless services are made available through dedicated apps. Password vaults like KeePass and Keeper, are another popular strategy. They typically transfer a password by having the user copy it, then paste it into the target app. This switching back and forth between apps is harmful to usability. The security impact is even worse, as the Android clipboard is publicly visible to all apps. Moreover, the Android clipboard provides a callback that notifies listening apps -including malicious ones- of any changes [14]. We also observe that many password managers implement their own secure storage, rather than relying on

platform facilities [2], [14]. A last client-based approach is the image-based ObPwd [12]. Provided as an Android app and as a Firefox add-on, it constructs textual passwords from user-selected objects, typically pictures. ObPwd relies on the assumption that these remain private. Hence, careless management, loss or theft of these objects, can jeopardise derived passwords.

The third class of approaches are deployable to the mobile, at the authentication provider or on both. The focus is to increase password usability by making it easier to enter and remember them. Jakobsson et al propose *fastwords* [11]. These are compound passphrases: the user can freely combine simple words, with the exception of obvious or popular combinations. Not only are they easier to remember, they also take advantage of the predictive typing and auto-correction in modern mobile keyboards. Bicakci et al present *gridWord* [3], a two-dimensional grid where the user selects the correct word out of a predetermined list, then positions it in the correct grid cell. This sequence is repeated twice. Because gridWord leverages the user's spatial memory, it is more usable than passwords. The question for fastwords and gridWord is whether they will suffice to help retain the numerous passwords that users encounter nowadays. Also, depending on the implementation, changes to existing systems may be required.

III. Approach

The storage and use of passwords are the two essential functions of a password manager. A user can request its services, from within a native app or from a Web application that runs in the mobile browser. This section first focuses on the native case, then extends it to Web applications. The description is intentionally kept conceptual, to overcome platform specificities. Our approach, a dedicated app, consists of two components. The password management keyboard provides password provisioning and standard input functionality. It is backed by a password store, which is responsible for securely storing user names and passwords.

A *relying app* prompts the *user* to enter his user name and password. We assume they have been previously stored. The *keyboard* is now initiated by the user, upon which it retrieves the identifier of the app via the *OS*. The *password store* is then queried for the user name and password that correspond to this id. Since they are maintained in encrypted form, the password store first retrieves its symmetric key from the *system keystore*. The keyboard thus receives the decrypted user name and password from the password store. Subsequently, the user is prompted to indicate the user name field. He pinpoints its location (e.g. by tapping it). This causes it to gain focus, allowing the keyboard to populate it. The password is transferred to its corresponding field in the same way. A similar workflow is followed to store or update a user name and password. This particular approach, in which the relying app's fields must have focus for the keyboard to read or populate them, is required. Mobile platforms impose strict security on input methods (e.g. keyboards). For Android, this is further detailed in section IV-A.

Two extensions are added to the above use case. First, the workflow becomes slightly more intricate if a password provisioning request originates from within a browser. In that case, passwords are not only associated with the relying app, but also with the Web application's URL domain. Consequently, the user is also prompted to tap the address bar. The URL is then converted to a canonical form. Queries to retrieve a password, take place by app id *and* by domain. In a second extension, multiple passwords per relying app are supported: a selection list is presented to the user, prior to populating the fields.

IV. Prototype

The concepts from section III are validated using an Android prototype. A single app encompasses both the keyboard service and the password store.

A. The keyboard and the Input Method Framework

The keyboard implementation is realised by extending `InputMethodService`, a service that Android specifically provides for new input types. These are not necessarily keyboards. Nothing precludes the use of voice or handwriting recognition. This service is part of our dedicated prototype app.

`InputMethodService` is one of the three components that make up Android's *input method framework* (IMF) architecture [1]. One or more *input methods* (IMEs) implement an interaction model that allows the user to generate text. The preferred IME can be selected in the system settings or via the notification tray. Android applications behave as *input clients* of these input methods. The interaction between both is controlled by the *input method manager*. The framework imposes strict security measures. First, only Android has direct access to an input method. As such, it mediates the communication between IMEs and clients. The system designates one input client at a time as *active*. Only this client may communicate -through the IMF- with the user's preferred input method: calls of inactive clients are ignored. Hence, at any given moment, only one input client and one IME are communicating. Lastly, clients are not allowed to programmatically switch to another IME. Instead, they can only request the user to pick a different IME. This prevents malicious apps from acquiring the capability to eavesdrop on the user's keystrokes.

B. Password store

Depending on the requirements and the existing infrastructure, the password store can be implemented in different ways. Robust protection against offline attacks can be attained using hardware-backed crypto. One can use a tamperproof module to implement this. However, this often incurs additional hardware cost per user. Instead we can either assume an adversary with limited computational capabilities for offline attacks or we can use a device that has hardware-backed credential storage. More such devices have been appearing since the introduction of the KeyChain in Android 4. As of Android 4.3, calls are available

to verify if hardware-backed storage is present and if private keys of a given cryptographic algorithm are hardware-backed and, therefore, non-exportable [15], [5]. Our prototype follows this last approach: it is implemented on a Nexus 4 device. As a result, the password store is well-integrated into the platform's MDM facilities. Since users often have multiple smartphones or tablets, backups and cross-device synchronisation are a non-negligible concern. Note that backups need long, high complexity passwords, since they are not protected by crypto hardware.

For local password storage, it suffices to use a symmetric key that is generated at install-time and kept in the Android keystore. Currently, however, hardware backing is only available for some asymmetric algorithms[1]. To overcome this limitation, the symmetric key for password storage is wrapped with a hardware-backed public RSA key, also generated during installation. An interesting advantage to this approach is discussed as an extension in section V-C.

V. Evaluation

A. Security

In our prototype, the keyboard interacts with the relying app through the Input Method Framework. Therefore no sensitive information is leaked via Intents or through the clipboard. If a device is stolen or lost, the screen and the password store are inaccessible without the user's passcode. Furthermore, the encryption of the storage mechanism is hardware-backed, which hinders offline attacks. Due to how passwords are stored and provisioned, our approach performs better than commercial solutions [2], [14]. Lastly, characters are not typed individually, they are imported in bulk. This makes our solution more resilient to shoulder surfing. In the remainder of this section, the anticipated attacks are listed, along with how they are prevented or mitigated by our approach.

1) Protection against malware:

M_1 **Interception of communication between the keyboard and the relying app.** The Input Method Framework prevents this attack by enforcing that only the active app can bind to and communicate with the current input method (see section IV-A).

M_2 **Impersonation of the relying app.** Android enforces its sandbox by typically assigning a unique Linux `uid` to every app. The keyboard, which extends `InputMethodService`, can retrieve it from the current input binding. Android permission enforcement is largely based on it. Hence, impersonating an Android app would require malware to sidestep platform security. However, the assigned app uid is different across devices, which would hinder password synchronisation between the same app on these devices. Therefore, we use the package name[2], which is obtained through an extra translation step. This approach results in a another, less likely attack. A malicious app spoofing a package name, can have itself installed before a legitimate app, f.i. when a backup is restored. This is prevented by also keeping track of the application signature. The combination of the package name and the signature are infeasible to forge by malware. Note that the package name cannot be omitted, as it is used to distinguish between different apps of the same developer.

M_3 **Interception of user input.** A possible way to capture user input, is for malware to (partially) cover the relying app and record the user's motion events. Android provides a flag `FLAG_WINDOW_IS_OBSCURED` to verify whether or not the window receiving the event, is on top. One attack in literature [17] uses the device's motion sensors to infer screen keystrokes. This requires a considerable level of sophistication. Moreover, the applicability of our approach stretches beyond keyboards, to input methods in general.

M_4 **Stealing passwords from the password store.** Since the password store is part of the password management app, it is isolated from other apps by the Android sandbox. The key material that it uses for encryption, resides in the Android keystore and is only accessible by the password management app itself.

2) Protection against theft:

T_1 **Installing a malicious app.** This attack is prevented by the same countermeasures that protect against malware. Furthermore, the user's passcode is required to unlock the screen.

T_2 **Rooting the device *after* obtaining it.** Root access can be obtained by unlocking the bootloader in supported devices and injecting a root-privileged app in Android. However, this also wipes user data, including the stored passwords. A way around this is to exploit an unpatched Android vulnerability. This potentially allows to circumvent OS security controls and to access the entire storage. However, the keystore is hardware-backed and the user's passcode is still required to access its contents.

T_3 **Offline attacks on the password store.** The encryption of the password store depends both on a user secret and on hardware. Hence, this entails carrying out brute-force attacks against the cryptographic algorithm.

Increased protection can be attained by adding password and screen lock policies. In a corporate environment, these can be pushed by an MDM server. A developer can also impose his policies, by leveraging the Device Administration API. Furthermore, user-selectable measures can be offered, which Android can execute if a condition is (not) met. As an example, our prototype provides application-level wiping: stored passwords and the keys protecting them, are removed from the device after a maximum number of failed password attempts. This last measure, however, is not resistant to rooting.

B. Usability

The framework by Bonneau et al [4] contains 8 usability criteria, which our approach fulfills. It is quasi-memorywise

[1]Developers can verify if a key is hardware-backed using `KeyChain.isBoundKeyAlgorithm("[algorithm]")`

[2]E.g. *com.google.android.gm* for Gmail

effortless, scaling to the number of passwords. No additional items must be carried. The physical effort is limited: three screen taps, which can be reduced to one, through platform support and standardisation of field naming. This yields a high efficiency of use. There is a smooth learning curve, due to simple, self-explanatory actions. Apart from initial storage, passwords are no longer entered, accounting for less errors. Lastly, recovery from loss is possible through backup and synchronisation.

A more approach-specific advantage is that there is no switching back and forth between apps, contrary to the popular mobile vault apps.

C. Extensions

In the initial concept, passwords are stored separately for each relying app. Scenarios exist where applications may benefit from using a common pool of passwords. Browsers are a good example. A configuration interface can be implemented, where the user initiates a password pool and adds relying apps to it. A complementary approach not depending on the user, is to automatically maintain such a pool for apps that have a common developer signature. For instance, these apps could be part of application suites that use the same authentication infrastructure.

Another addition is of value to corporate use. The organisation can impose restrictions on apps that are allowed to store and retrieve passwords. Application white- or blacklisting can be incorporated in the organisation's MDM policies.

Section IV-B proposes to wrap the password storage key with a public key, prior to saving it. This overcomes the limitation that symmetric keys are not hardware-backed. This approach allows device-bound synchronisation. If a user has multiple Android devices, on each of which the password management app is used, every instance has a hardware-backed private key and a corresponding public key. In a pairing phase, each device exchanges public keys with the others. To synchronise stored passwords remotely, e.g. via an untrusted cloud, a second symmetric encryption key is generated. The device sharing the passwords, encrypts them with this key. Wrapped versions of the key are generated, one with each public key. Finally the encrypted passwords and the wrapped keys are uploaded to the untrusted cloud. As a result, the passwords are only accessible to the participating devices, since hardware-backed keys are non-transferable.

D. Limitations

Android is currently the only mobile platform that allows replacing the default keyboard. On Blackberry, a third party keyboard[3] has been the system default since version 10. We reasonably expect our approach to be portable to future versions of other platforms. Essentially three API functionalities must be available. First, access to the identifier of the relying app is required. Moreover, we need read and write access to a text field that has focus, a trivial requirement for a keyboard. Third, the input method must be made aware when a field gains focus, in addition to which field this is.

A second limitation, is that this approach only supports password-based credentials. Since the absolute majority of authentications still relies on passwords, this limitation has little impact.

[3]http://www.swiftkey.net

VI. Conclusions

This paper has proposed a mobile password management keyboard that can accommodate any Android application. Our approach leverages platform facilities to better protect against malware and device theft than existing solutions. In comparison to mobile vault apps, passwords are more securely stored and provisioned. In addition, the user's workflow is minimally disrupted, as core functionality is directly available from the keyboard. Our approach is not limited to the passwords of a single application, in contrast to browser-based solutions. Ultimately, these improvements lower the barrier to working with stronger, more differentiated passwords. Neither the mobile OS nor the relying apps need modification. The proposed approach has been validated by a prototype.

References

[1] InputMethodManager | Android Developers. https://developer.android.com/reference/android/view/inputmethod/InputMethodManager.html, September 2013.

[2] Andrey Belenko and Dmitry Sklyarov. "secure password managers" and "military-grade encryption" on smartphones: Oh, really? Technical report, Elcomsoft, Amsterdam, March 2012.

[3] Kemal Bicakci and Paul C. van Oorschot. A multi-word password proposal (gridword) and exploring questions about science in security research and usable security evaluation. NSPW 2011, pages 25–36.

[4] J. Bonneau, C. Herley, P.C. van Oorschot, and F. Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 553–567, 2012.

[5] Nikolay Elenkov. Credential storage enhancements in android 4.3. http://nelenkov.blogspot.be/2013/08/credential-storage-enhancements-android-43.html, August 2013.

[6] Cormac Herley et al. A research agenda acknowledging the persistence of passwords. *IEEE Security and Privacy*, 10(1):28–36, January 2012.

[7] David Recordon et al. OpenID 2.0: a platform for user-centric identity management. In *Proceedings of DIM 2006*, pages 11–16.

[8] Denise Ranghetti Pilar et al. Passwords usage and human memory limitations: A survey across age and educational background. *PLoS ONE*, 7(12):e51067, 12 2012.

[9] Jeff Yan et al. Password memorability and security: Empirical results. *IEEE Security and Privacy*, 2(5):25–31, September 2004.

[10] Markus Jakobsson et al. Implicit authentication for mobile devices. HotSec 2009, page 9. USENIX Association.

[11] Markus Jakobsson et al. Rethinking passwords to adapt to constrained keyboards. In *Workshop on Mobile Security Technologies*, 2012.

[12] Mohammad Mannan et al. Passwords for both mobile and desktop computers: ObPwd for firefox and android. In *;login: issue: August 2012, Volume 37, Number 4*, 2012.

[13] R. L. Morgan et al. Federated security : The shibboleth approach. *EDUCAUSE Quarterly*, 27(4), 2004.

[14] Sasha Fahl et al. Hey, you, get off of my clipboard. In *Proceedings of Financial Cryptography and Data Security 2013*.

[15] Google. Android 4.3 APIs - android developers. https://developer.android.com/about/versions/android-4.3.html#Security, August 2013.

[16] M.Z. Jali, S.M. Furnell, and P.S. Dowland. Assessing image-based authentication techniques in a web-based environment. *Information Management 38; Computer Security*, 18(1):43–53, 2010.

[17] Zhi Xu, Kun Bai, and Sencun Zhu. Taplogger: inferring user inputs on smartphone touchscreens using on-board motion sensors. WISEC 2012, pages 113–124.