Crazy Secure Protection for Android Monero Wallets

m2049r

March 19, 2018

Abstract

In general, small mobile devices provide cumbersome input methods as well as limited screen resources. This leads users to create weak passwords & passphrases to secure even the most sensitive of information. This white paper presents a solution for securing Monero wallets on Android devices regardless of where the files are stored. The procedure involves the user being encouraged to enter a strong passphrase whose RSA signature is hashed with the CryptoNight algorithm to produce a 256-bit key which can be used as the real wallet passphrase. This procedure ensures that even weak passphrases generate strong 256-bit based passphrases.

1 Introduction

In general, small mobile devices provide cumbersome input methods as well as limited screen resources. This leads users to create weak passwords & passphrases or even simple PIN-codes to secure even the most sensitive of information.

One possibility of resolving this problem is by educating the user about security and passphrase selection through help dialogs or popups. This can be enhanced further through the use of nagging messages during passphrase creation by quantifying the strength of the entered text and displaying this info either in text form or as a progress bar. A more drastic approach is to enforce certain rules on password selection.

The efficacy of passphrase rules is controversial and a negative user experience. The constraints of small mobile devices also need to be considered in this regard, where often changing from letters to digits or special characters is tedious and the space provided on the screen for the passphrase is limited.

In the "mobile wallet scenario" a passphrase is used to secure the private keys of the user. These are stored in an encrypted file on the device. It is often proposed to store this file in internal file storage because only the app can access its files in this storage area. The reasoning behind this approach is that no other (potentially malicious) app can access this area and thus the files are safe. A weak password is good enough as the files cannot be accessed. Although this is a better approach than storing wallet files in external file storage where any other app can access the files, as soon as the wallet files are exported (for example to create backups, migrate to new device, etc.) and so released into the wild, they are prone to the usual attack vectors. A malicious app just needs to wait for this to happen. An often overlooked attack vector is that the app itself can be easily modified and so has access to all data the original app has access to, including wallet files.

We propose a slightly different approach.

1.1 Assumptions & Attack vectors

This paper is based on the implementation of Monerujo – a Monero wallet for Android.

We can only control what we can control. Installed trojans like Svpeng pose attack vectors we are not aiming to resolve. It grants itself device administrator rights and can log keystrokes by use of the Accessibility Services. If the password screen is protected against screenshots, it also displays a screen overlay which looks like original screen and logs entered passwords even if the app does not use standard keyboard entry. This we cannot control.

What we want to control is the strength of the passphrase used for the encrypted wallet files, so that these can be "shared freely" (be it on or off the device) without fear of being opened by a third party. We want to mitigate attackers stealing mobile wallets and guessing the encryption keys. As we don't think that stealing of files from an Android device can be effectively mitigated, we are concentrating on producing strong encryption keys.

For our solution we use RSA encryption by way of the Android Keystore System which protects key material from unauthorized use. In particular it mitigates unauthorized use of key material outside of the Android device by preventing extraction of the key material from application processes and from the Android device as a whole. This system is more secure than the aforementioned internal file storage because even the app itself cannot retrieve the encryption keys. In addition, the key material never enters the application process. When an application performs cryptographic operations, the data is fed to a system process which carries out the cryptographic operations. If the app's process is compromised, the attacker may be able to use the app's keys but will not be able to extract their key material (for example, to be used outside of the Android device). Key material may be bound to the secure hardware (for example, Trusted Execution Environment (TEE),

Secure Element (SE)) if supported by the Android device hardware. When this feature is enabled for a key, its key material is never exposed outside of the secure hardware.

In the case that an attacker gets access to the key files, we also want a bruteforce attack on the keys to be costly and slow. Our approach thus also requires a slow hashing function. As we are working in the Monero-Space, we have selected the Monero KDF (CryptoNight) to be used as this slow hashing function.

2 Materials & Methods

We propose passphrase generation for the wallet files be done in 5 steps, the first of which need only be executed once per app installation:

0. Generate a 2048-bit RSA keypair for PKCS#1 signing operations only in the Android Keystore System and reuse it for all wallets.

1. The user is asked to enter a strong password. The password strength is measured during entry and displayed in the form of nag-messages until a certain strength is reached. The passphrase assessment is done by the zxcvbn algorithm. A passphrase is deemed as secure enough if the algorithm estimates that more than 10^{13} guesses are required to crack it.

2. Sign the user-passphrase (using UTF-8 decoding to cater for different languages and character sets) with the secret RSA key through the Android Keystore System.

3. Hash the resulting signature (256 bytes) with the Monero-inherent CryptoNight algorithm to produce a 256-bit key.

4. Convert the 256-bit key into a mnemonic the user can write down for further use using the Monero mnemonic-derivation algorithm.

3 Results & Discussion

What we are effectively doing is creating a crazy secure passphrase for the user.

We are using the RSA signature as "salt" for the user-passphrase. We could also simply generate a 32-bit random salt which we can store in the app's shared preferences which we prepend to the user-passphrase before CryptoNight hashing. But the shared preferences could be targeted by an attacker with relatively little effort. In that case, we could encrypt the salt with our RSA keys (which are safely stored by the Android Keystore System) and store the encrypted salt in the shared preferences and decode it when we need it. Although this seems to be a popular approach, an attacker who manages to hack the app can easily decode the salt and use it to bruteforce the wallet passphrase off-device. Our approach forces the attacker to bruteforce on-device in such a scenario – if they do not manage to get access to the secure Android Keystore System. This is the reason we have opted for using the RSA signature as salt.

3.1 Implications for attackers

If an attacker steals the wallet files only, they would have to guess a 256-bit passphrase. This is as secure as the Monero seed itself.

An attacker may manage to not only steal the wallet files but also to compromise the app. As the RSA key material is securely held by the Android System, any bruteforce attack on the user-passphrase would have to be done on the device itself as the cryptographic operations can only be performed on the device by the app itself. A Galaxy S4 Active (ARM32 processor @ 1.9GHz) generates our crazy secure passphrases at a rate of only 3 per second.

If the attacker manages to steal everything from the device, including the RSA key material, they will be able to perform a bruteforce attack on dedicated hardware. If the user honoured the suggestions during user-passphrase creation in step 1, their passphrase would require more than 10^{13} guesses to crack. It should be considered that the key material may be on dedicated

hardware if the Android device provides it and so this attack is avoided altogether (within limits).

3.2 Variations

RSA was chosen because Monerujo support pre API 23 devices and those only allow RSA keys to be used with the Android Keystore System. On post API 23 many more cryptographic options exist.

One variation worth noting, is the use of a seperate RSA key for each wallet. The open question here is if and when to delete keys which no longer correspond to wallets on the device.

3.3 UX/UI

The user must be made aware of the intricacies of this approach through cunning UX/UI design. Although we assume, that the majority of users do not share wallet files among devices, they should be making backups which they can deploy on new or factory-reset devices. Thus it is important for them to store the 25-word passphrase generated by the proposed approach alongside their mnemonic seed. It is important for the user to understand the difference between the two. It may be wise to reduce the generated wallet password to 13 words mymonero-style or use some other format which is clearly different from the Monero mnemonic seed.

Exporting of wallets should be accompanied by a message providing information about where to find the actual passphrase of the wallet. A similar message needs to be presented on wallet creation and on change of passphrase operations.

Importing of wallets with a crazy secure 25 word passphrase should offer the user the possibility to regenerate a new passphrase according to the methodology proposed. This would, as a consequence, also change the real passphrase of the wallet.

To cater for old-style wallet passphrases, the entered passphrase needs to be checked against the wallet file directly,

and if this fails, go through the described procedure to derive the secure password.

Old-style passphrases can be converted to crazy secure passphrases easily. The question is, if a user should do this as the wallet may already be compromised. If this is a possibility, it is certainly better practice to create a new wallet and transfer all funds.

On uninstallation of the app, the keystore is wiped together with the internal file storage. The user should be made aware of this *before* this happens, as there is no possibility for the app to react when it does.

4 Acknowledgments

#monero-research-lab especially hyc, luigi1111w, moneromoo, sarang and suraeNoether for their feedback.

5 Bibliography

zxcvbn: Low-Budget Password Strength Estimation, https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/wheeler

Java port of zxcvbn, **https://github.com/nulab/zxcvbn4j**

Android Keystore System, https://developer.android.com/training/articles/keystore.html

Dangerous Mobile Banking Trojan Gets 'Keylogger' to Steal Everything, https://thehackernews.com/2017/07/android-banking-malware.html

TrustZone Downgrade Attack Opens Android Devices to Old Vulnerabilities, https://www.bleepingcomputer.com/news/security/trustzone-downgrade-attack-opens-android-devices-to-old-vulnerabilities/