Bachelor thesis
Computer Science

Radboud University

Protecting app secrets in Android

Author:
Matjah Sonneveld
s4121325

First supervisor/assessor:
Fabian van den Broek, MSc
f.vandenbroek@cs.ru.nl

[Second supervisor:]
Sietse Ringers, MSc
s.ringers@rug.nl

Second assessor:
title, name
e-mail adress

July 5, 2016

Abstract

In this paper we will look at ways to protect a secret key present in an Android application. This could be anything a developer would want to keep secret such as an API key but we will look at a key used in the app of the IRMA project. This project gives us some restrictions that do not allow us to use the standard Android solutions created for such problems. We will look at existing alternatives and come up with our own solution that combines these alternatives. In the end we could not find a solution that protects us from all kinds of attackers but we do make it significantly harder to retrieve and use our secret key.

Contents

1 Introduction **3**

2 Preliminaries **4**
2.1 IRMA 4
2.2 The IRMA App 4
2.3 Attacker Models 5
2.3.1 Hostile Applications 5
2.3.2 Stolen devices 5
2.3.3 Rooted devices 6
2.4 Server-side secrets 6

3 Research **7**
3.1 Android keystore 7
3.2 Proguard 8
3.3 Android Native Developmentkit 8
3.4 Opensource vs Security 9
3.5 User password 9
3.6 Device secret 10
3.7 Password Hashing 10
3.8 Encryption 10
3.9 Storage 11
3.10 Usage of our encryption class 11
3.11 The Algorithm 11

4 Proof of Concept **14**

5 Related Work **16**

6 Further Work **17**

7 Discussion **18**
7.1 Applications 18
7.2 Stolen devices 19
7.3 Rooted devices 19

7.4 Users 19

8 Conclusions **20**

Chapter 1

Introduction

This research is about protecting app data in Android apps, in particular the secret key used in the Android app written for the IRMA project[1]. The solution in this paper is however not limited to the IRMA project, it could be used in any app that wants to hide any form of key (for example a cryptographic or API key). In the case of IRMA, if the secret key would be extracted from the app it could be used, together with the stored attributes, to transfer those attributes to another account. This could lead to people having the over 18 attribute while they are actually 17 years old. At the moment the key is a variable in the app and would be easily extracted by doing a memory dump while the app is running or by looking at the IRMA app storage. With this research we focus on app's that want to protect information that cannot be protected using the standard practices for Android which gives us restrictions on which standard security measures we can apply. Because our main focus is the IRMA app we have to deal with some restrictions that prevent us from using security measures that would be used otherwise. For example, a good method to protect the key would be to encrypt it using the Android Hardware-Backed Keystore[2]. However the amount of devices having this hardware encryption enabled is estimated to be as low as 33% [9] while it is the aim of IRMA that the app should be usable on all Android devices. Because of this we will look for a solution that is software based and thus compatible with every Android device. There are also other reasons for us not using the Android Keystore which will be explained in the preliminaries. In this paper we will look a at which threats we face when dealing with these restrictions. We will research alternative existing methods and combining these into our own method which protects the app data from attackers. Finally we will implement this solution into a proof-of-concept that shows our method could be used in real world apps.

[1] `https://www.irmacard.org/`
[2] `https://source.android.com/security/keystore/`

Chapter 2

Preliminaries

2.1 IRMA

The IRMA project (short for I Reveal My Attributes) is a project that offers privacy friendly authentication. This means you authenticate with a subset of your attributes instead of revealing all of your attributes. When you have to verify one of your attributes like name, age or nationality to a liquor store you should not have to reveal all of your other attributes as well, which is the case if you have to show you identity card or passport. This is where IRMA is different as it gives you the choice of what attributes you want to share, in the liquor store example you would only show your age and not your name or nationality. The exact details of this project can be found on the IRMA website[1].

2.2 The IRMA App

The IRMA app is one of the ways to store your attributes and sharing them to whomever needs to verify them. One of the ways to obtain a set of attributes is to read the RFID chip of a document like a passport or drivers licence with an Android phone. Because the IRMA app does a lot of internal cryptography operations that are not relevant for this research we will create a small model that we can use for further reference. We will define a function $E_{key}(challenge)$ where E is the internal cryptography of IRMA, *key* is the secret key we want to protect and *challenge* is the input in the form of a cryptographic challenge to which we need to give correct response. The exact content of E can differ for different actions such as issuing and revealing an attribute but for simplicity we will just use E. In the general case of another app wanting to protect its data this E could be any function that app uses to verify or use that data.

[1] `https://www.irmacard.org/irma/`

2.3 Attacker Models

There are three different scenarios that we want to protect against. We will discuss each one briefly. In these models we will describe how they would affect our IRMA example where the attacker is after our *key* but each of these also apply to the general case of hiding any data inside an Android app (for example: an API key or user information). In each of these models we assume the attacker is not trying to just use the app which would mean to simply obtain a valid response to a challenge ($E_{key}(challenge)$).

2.3.1 Hostile Applications

The first scenario would be a hostile application installed on a user's device. This could be a app that tries to gather user data from other installed apps, including our own IRMA app. We will assume the app does not have root access and is not specifically targeted at the IRMA app. To defend against this threat we will have to store our key in a place that other apps cant reach. Because we assume that the app has no root access we can trust in the Android OS and its file system and permissions.

2.3.2 Stolen devices

The second scenario is that a user's device gets stolen. For this scenario we assume the attacker is not able to root the device, if he is able to do so this is covered in the Rooted devices scenario. A big difference between this attacker model and the other attacker models is that in this a the key is not used by the app while the attacker is active which means it does not need to be in a decrypted state. If the stolen device does not have password protection or does not have disk encryption enabled the attacker could easily root the device. We only have to protect our key while it is stored on the file system. One possible motive for stealing a device would be that the attacker wants to retrieve the user's secret key so that he could then use the attributes of the user. We would have to make sure that the data again is in a hard to reach place and that it is not possible to extract it from a locked device. In most cases the OS would prevent anyone from copying data of a locked device but if Developer Debug Mode is enabled it is easy to use recovery software that can copy any file from the password locked device[2]. An even better way would be to encrypt the key to make it unusable outside of the app. This second option should be enough on its own but making the key harder to extract is a good layer of extra protection.

[2] `http://www.file-recovery.com/android_recovery.htm`

2.3.3 Rooted devices

A third and possibly the most dangerous threat is a attacker that is an IRMA user. It is possible that he would want to retrieve his own secret key, this would make it possible to, for example, sell his attributes to others. What makes this model dangerous is that this attacker could easily have rooted his device meaning he has root access to the operating system. Completely defending against this is nearly impossible because the attacker has full access to the file system and has the possibility to make memory dumps at each given moment. What we can do however is make sure that the secret is never on the file system in plain-text and make the opportunity window for a memory dump as small as possible. The goal of this research is not to make the secret impossible to retrieve but to make is as hard as we possible can for an attacker. In the Discussion we will look at these models again to see if and how we are able to protect ourselves against these threats.

2.4 Server-side secrets

A very obvious solution to the problem of protecting secrets on a client application is to move it to a server. This makes an attack harder because an attacker has to break into your server instead of just having an app to play around with on his rooted Android phone. There are however a few reasons for us to not use this approach. First of all, this goes against a key idea of the IRMA project which is that you should have full control over your own attributes and that the system should be fully distributed. Another reason is that if you move the secret to the server you would still have to authenticate to the server. To authenticate you need credentials and that creates a new problem of securely storing those. All in all storing the secret server-side seems like a good solution for many applications but in our specific case of the IRMA app it does not fit. Therefore we will find our own method of solving this problem.

Chapter 3

Research

In this chapter we will look into the research we have done. The first sections are about researching known methods to help protect app data and in the last sections we will come up with our own combination of these methods.

3.1 Android keystore

A obvious choice to store a secret key on Android would be to make use of its integrated Keystore that is available from revision 18 of the Android API[1]. The keystore is designed to store cryptographic keys in a container which would then be difficult to extract from the device. How this is done depends on the device hardware. In our case we do not want to depend on the presence of certain hardware because we want the app to be usable on all devices as described in the introduction. When this hardware is absent the Keystore falls back on its software implementation. When the device requires a PIN to unlock it, this PIN is used to derive a masterkey which is then used to encrypt the keys in the Keystore. A paper that analyses storage on Android including the software fallback of the Keystore [10] concludes that the Android Keystore does indeed make it more difficult for an attacker to retrieve keys from the Keystore, but does not make it impossible. For our own solution we also know that it is probably impossible to come up with a completely secure way of storing our own key but we want to make it as difficult as possible. An important reason to not use the Android Keystore is that our application uses very specific encryption and decryption function E_{key} is not in the API. This means that if we store the key encrypted in the Keystore, each time we want to use the key for our function E, we have to decrypt it and move it to unsecured memory, removing the most important strength of the Android Keystore. Last but not least there are some documented cases [6] of the Android Keystore 'forgetting' all its stored

[1] `http://developer.android.com/training/articles/keystore.html`

information when the user changes it PIN or authentication method. This could be explained by the fact that the master key is derived from this PIN as described earlier. If this happens our IRMA app will be essentially wiped of its attributes because we can no longer respond to any cryptographic challenges.

3.2 Proguard

When looking for recommendations to hide and protect secrets in Android apps one of the first suggestions often is Proguard [8]. Proguard is a tool that among other things re-factors code to make it more compact because it renames method, variable and class names to make them shorter[2]. Proguard does this by first figuring out which labels it can rename. It then chooses new names for these fields that have no semantic meaning. After the renaming it checks for conflicts and then fixes these conflicts. Because the renamed labels no longer have a semantic meaning it is a lot harder to reverse engineer it. This wont hide any secrets but it will be harder to find them. Because Proguard is integrated into Android we only need to set the minifyEnabled setting to true in our Gradle build file.

3.3 Android Native Developmentkit

Another suggested way of making reverse engineering harder is by using the Android Native Developmentkit (NDK). The NDK allows for code to be written in C/C++. This code is harder to decompile, while there are single click applications that can decompile Java code, decompiling C++ requires expert knowledge and time [7]. There are also obfuscation and tamper-dectection tools available to make decompiling NDK code even more time consuming such as the Obfucator-LLVM project [3]. This gives us a place where we could write a library that can encrypt and decrypt our key without someone just reversing the encryption process. In the end this can be considered Security through Obscurity[3] which is of course not something considered good security but is does protect from a fair amount of non-experienced attackers.

Memory management is another issue with Java in Android. We would like to delete the decrypted key from the memory as soon as were done with it. In C++ we have direct access to the memory so we can just overwrite or deallocate the bytes of memory that hold our secret key while in Java we can only dereference the key and hope that the garbage collector cleans it up before the key is stolen making the opportunity windows smaller. A

[2] `http://developer.android.com/tools/help/proguard.html`

[3] `https://en.wikipedia.org/wiki/Security_through_obscurity`

disadvantage of using the NDK is that writing native code is often risky. It is easy to make mistakes that could lead to decreased performance or even memory leaks. Another disadvantage is that NDK projects need to be compiled for every chipset the app is intended to run on. This gets increasingly troublesome when including external libraries. Extra compiled libraries also increase the final apk size of applications which is not something we want.

3.4 Opensource vs Security

Our main focus in this research is the IRMA project. This project, including the Android app, is a open-source project. This means that all of our source code should be open for anyone to inspect and download to use for themselves. The decompiling issues we have talked about in the NDK section are of course meaningless if the source code is publicly available. Open-source is a great concept but it does have a big disadvantage when you want to keep something secret. Luckily we decided in the previous section that it would be a good idea to place our encrypting and decrypting in a separate NDK library. This means that we can make this a kind of interface that stays closed-source. Anyone can still download the source code of our main Android application but has to provide his own library that replaces our version. Alternatively a compiled binary could be shipped with the code. It could even be implemented in the code that if there is no NDK library present the app should default to using no encryption, that should make for less hassle but could be dangerous if people are not aware of the lack of security.

3.5 User password

To provide an extra layer of security it is a good idea to make use of a piece of information that the user has to input each time he or she wants to unlock our secret. This could be useful if the device gets stolen and rooted. An attacker could patch our app and try to call our decryption function to simply retrieve the original secret key. We will prevent this by asking the user for a password the first time we generate and encrypt the key and each time we want to decrypt the key. Note that this could be a text-based password or a form of biometric scan (like a fingerprint) as long as it appears a some kind of string in our app. This string could then be used as an encryption key to encrypt our secret.

3.6 Device secret

In the previous section we tied the key to our user but we also want to tie the key to our device. To achieve this we have to take a device specific string like a serial number or the IMEI[4]. Both are device unique numbers that can be requested by any app and used in its code. If we incorporate one of these string in the encryption of our secret key this would protect against an attacker extracting the key from the device and cracking it on a more powerful machine. This is a weak protection mechanism when dealing with professionals because serial- and IMEI numbers can be copied from the device when the attacker knows he needs them. Because IMEI numbers can be changed when an attacker has root access we will choose the device serial in our solution.

3.7 Password Hashing

An example of a password hashing algorithm in PBKDF2 [4]. This is an algorithm that produces a derived key from a password together with a salt. The main purpose of this algorithm is to stretch a user given password to a much larger string which then can be used in other cryptographic operations. A big advantage of doing this extra work is that it adds computation time which makes it harder to brute force passwords using rainbowtable or dictionary attacks. We picked PBKDF2 because it is a widely used algorithm that is supported in most crypto libraries. In our case we will use it to hash our password with the device serial number as salt. Adding a salt has the benefit of also protecting against generic pre-computing attacks.

3.8 Encryption

Encryption a key had the benefit of the key being useless in its encrypted form. An attacker could obtain the key but it will be useless until it is decrypted. This way we have to worry less about attackers extracting the key from our app. To actually encrypt our secret key we will use AES. We will use the 256 bit variant with CBC cipher mode because its commonly used, which will help us when creating our proof-of-concept, and generally considered secure. [5] The input of AES-256 will be the secret key. As the 256-bit encryption key we will use the user password hashed to a 256-bit string with the device serial number as salt. The result of $PBKDF2(user\ password, device\ secret)$ will be called *unlock_key* for further reference.

[4] `https://en.wikipedia.org/wiki/International_Mobile_Station_Equipment_Identity`

3.9 Storage

After we encrypt the secret we still need a way to store it. We don't want other apps to have access to the key so we cannot use external storage. It is of course true that an encrypted secret is secure on its own but any extra layer of security is welcome. A much suggested place to store app specific and sensitive data which has no place in the Keystore are the shared preferences of an app [1]. Another reason for not using the Keystore for this key is again the 'Forgetfull Keystore' problem which would render all attributes in the IRMA app useless. Permissions are set in a way so that only the app's UID has access to this file. Against someone that has root access this of course is useless but it does keep other apps out. An app could be made to run under the same UID which would offset the security the filesystem permissions provide, however this would require a targeted attack against our app and we just want to protect against data gathering apps when it comes to picking a place to store our secret.

3.10 Usage of our encryption class

Now that we have an encryption class inside the NDK we have to make sure that the secret key never leaves the NDK. The $unlock\_key$ will be generated inside the NDK and just necessary output will leave the NDK (this ouput should, of course, never be the secret key). This is the case because a possible way an attacker could try to retrieve our key is to patch the app and log the output of our NDK library's decrypt function. An attacker could also choose to patch the Java side of our app to call the encrypt and decrypt function at will, which would output our encrypted secret. To prevent this we will add a function to our decryption method. In the case of the IRMA app these will be the $E_{key}(challenge)$ from our model. Functions like this would also have to be implemented on the NDK side. Each one of those functions should be called inside the decryption function and only the output of that function should be returned to the Java side of the app. That way the entire function is contained inside the NDK. For any other app besides IRMA those function could be replaced by any operation that does not output the secret itself.

3.11 The Algorithm

Combining all the previous ideas we have come to the following algorithm The whole of this algorithm is located in inside the NDK portion of our app and can be called from the Java side.

Encryption

This method has the user password as input parameter.

1. The user password is stored in temporary variable *pass*.

2. The device serial number is determined and stored in *serial_salt*. This should be done every time the algorithm runs because we want decryption to fail when the algorithm is run on a different device.

3. A password hashing algorithm is called (in this case PBKDF2). The input in set to *pass* and the salt is set to *serial_salt*. Because we want to make this algorithm slow to prevent brute forceing we will run multiple round of PBKDF2. This amount should be the largest number that is still tolerable performance wise [2]. Lastpass revealed in 2011 that it used 100,000 iterations so that is the number we will pick for now. The final result of 100,000 round of PBKDF2(*pass*, *serial_salt*) will be a 256-bit hash called *unlock_key*.

4. At this point the user password is no longer needed and should be removed from memory and will be zeroed.

5. Now we have our *unlock_key* we can call our encryption function. We will choose AES-256 with CBC cipher mode as described in our AES section. As input we pick our secret key (*key* from the IRMA model) and we use the previously generated *unlock_key* as encryption key. This will output our encrypted secret key_{enc}.

6. The original key *key* and *unlock_key* can now also be removed from memory. After this action the original *key* is no longer present on the device, we only have key_{enc}.

7. To make our secret key persistent we store key_{enc} in the Shared Preferences. This way we can access the secret if we closed and reopened the app.

Decryption

Each time the secret key is used this method should be called. This method has again the user password as input parameter

1. Again, the user password is stored in temporary variable *pass*. Note that this is required for each use of secret key. If this is too inconvenient we could choose to store this password until the app is closed or the device is locked.

2. Determine the device serial number and store it in *serial_salt*.

3. Password hash using PBKDF2 with *pass* as input and *serial_salt* as salt for the exact same number of iteration used in encryption. This results in *hash*.

4. At this point the user password is no longer needed and should be removed from memory and will be zeroed.

5. Retrieve key_{enc} from the Shared Preferences.

6. Do AES-256 with CBC cipher mode using key_{enc} as input and $hash$ as encryption key resulting in key, our decrypted secret key.

7. At this point we can perform the operation we want to do with key. In the case of IRMA this most commonly would be $R_{key}(challenge)$ but it could be anything. We will store any result of these operations in a variable $result$.

8. Now we will remove key from memory as it is no longer needed. All other temporary variables and resources used up to this point should also be freed except for $result$.

9. $result$ is returned to the Java side of the app.

Chapter 4

Proof of Concept

To see if we can implement our idea in a real android app we have made a proof-of-concept app. This is a simple android app consisting of of a simple GUI with a java side that interacts with a NDK library that does encryption and decryption as described in this paper.

To create our proof of concept app we used Android Studio 2.1 with its new experimental Gradle plugin for Android NDK[1]. This new version of Android studio comes with some code samples. One of these code samples is an Android app that retrieves a string from a C++ file and displays it on the screen. This project also includes two pre-compiled C++ libraries. We decided to use this project and replace the provided libraries with a pre-built version of OpenSSL[2]. This provides us with a working and compiling Android NDK app that has access to the whole OpenSSL library which includes AES and PBKDF2.

The example app consists of one Java file which makes a few calls to our NDK library. First it calls an encrypt method with a string that contains the users password. This encrypt method is a wrapper for the actual encrypt method that is based on the OpenSSL example[3]. This wrapper function converts the Java String to an unsigned char array, hashes this string with PKBDF2 and initialises the OpenSSL library. For this proof-of-concept we chose to hardcode the 128-bit IV used for AES-256 and the secret. In a real-world application the IV should be random and the secret should be initially set by the setup of the app and then immediately encrypted.

The second call of the NDK side is to the decrypt method with again the user's password as a parameter. The decrypt function is also a wrapper for a decrypt function based on OpenSSL's example. The decrypt wrapper converts and hashes the password in an identical way to the encrypt function.

[1] `http://tools.android.com/tech-docs/new-build-system/gradle-experimental`

[2] `https://github.com/emileb/OpenSSL-for-Android-Prebuilt`

[3] `https://wiki.openssl.org/index.php/EVP_Symmetric_Encryption_and_Decryption`

After that it initialises the OpenSSL library and decrypts our secret with the hashed password. The decrypted secret is then converted back to a Java String and returned.

Chapter 5

Related Work

Our particular subject is a problem encountered mainly by app designers and programmers. Curiously there does not seem to be many scientific sources that deal with similar problems. We have found a paper by Time Cooijmans, Joeri de Ruiter and Erik Poll [10] that does an analysis of the Android Keystore. This analysis proves the need for an alternative solution if we want to secure data on a phone that does not have support for hardware-backed cryptography.
The are however a lot of sources in the form of blog posts, stack-overflow questions and other online articles written by people looking for solutions of similar problems, for example the storage of API keys. A few examples of sources used in this paper are websites like Stack Overflow[1] which was a very good resource when we needed help in programming our proof-of-concept. Another example is `http://www.androidsecurity.guru` written by Simon Judge. This websites features lots of posts about Android security including a few directly related to our own problem.

[1] `http://stackoverflow.com/`

Chapter 6

Further Work

The proof-of-concept is very basic at this moment in time. A few options to look in to are the following:

- The algorithm provides a parameter for the number of iterations of PBKDF2 that should run. Ideally the number should be as high as possible while the delay is still tolerable for the user. When working with a platform that supports many devices with varying cpu power it is difficult to pick a number of iteration that does not result in unusable user experiences on slow devices while still begin effective in causing a long enough delay to make brute forcing difficult. A nice solution would be to benchmark a device on setup an determine the iteration count dynamically.
- Read/write to the Shared Preferences. Currently the key is written to a variable inside of the NDK class.
- Implement reference hashing- and encryption methods instead of using external crypto libraries. This would remove the need for compiling external libraries and reduce the final apk size.
- Research tamper-prevention possibilities[1]. This would allow detection of attackers trying to decompile or debug our code and help prevent these actions.

There are also many different password hashing algorithms besides PBKDF2. Recent alternatives include algorithms that use a set amount of memory to prevent using GPU-based hashing. But this could be difficult to implement given the many variants in devices

[1] `http://www.androidsecurity.guru/incorporate-tamper-detection/`

Chapter 7

Discussion

In this paper we looked at the possibilities of storing secret data on an Android device without using the Android Keystore. We have found numerous approaches to make it a bit harder for attackers to retrieve our secret. Among other things we found that moving our delicate code to the Android Native Developmentkit is a way of making reverse engineering harder, especially for those who are not experts in decompiling C/C++ code. We came across password hashing with PBKDF2 which makes it possible to use user input and hash it into a usable 256-bit string with the device serial number as salt. This makes it a lot harder for an attacker to extract the encrypted secret from a device and crack it with external specialised hardware. Finally we combined those things into an algorithm that uses the PBKDF2 hash as the encryption key for AES-256 and stores the encrypted key in the app's Shared Preferences.

In the preliminaries we defined three attacker models that we want to protect from. We will now revisit each of those and see what we have achieved.

7.1 Applications

This scenario assumed a hostile, information gathering application on the device that does not have root access. Because we now have our key in an encrypted form and placed it in the Shared Preferences of our app we have two layers of defence against such hostile applications. It can not access our key if we can trust the OS and even if it did it has no way of decrypting the key unless it also captured the *hash* generated by our algorithm. If the hostile app is able to make memory dumps there is the possibility of it capturing the key in its decrypted form in the small window of it being in memory while our decryption method is running. This however seems unlikely unless the hostile app was specially written to do so which is something we do not assume in this scenario.

7.2 Stolen devices

The second scenario is that the attacker has stolen a device containing the IRMA app. An attacker could want to extract the key to, for example, use it on another device. We can assume the app is not running (because the device is locked) and our decryption method will not be called and therefore our key is only present in encrypted form on the file system of the device. The attacker would first have to extract the key from the device, something that is possible with Android recovery programs which enables users to extract data when locked out of their device. The attacker still would have to decrypt the key. Because we moved our algorithm to the NDK it is hard to ascertain the precise steps of decryption. When an attacker has the expertise to decompile the C++ library he still has an 256-bit encryption key (the hash of our users password salted with the device serial key) to crack. Using external hardware to brute force this is not feasible.

7.3 Rooted devices

Thirdly we have the threat of an attacker that has root access. This attacker should have no problem retrieving the encrypted key but still has to perform the decryption as described in the previous subsection. A much bigger threat from this kind of attacker is that he is able to run our app at will and perform memory dumps along the way. While we have made an effort to make the opportunity window as small as possible the attacker will, given enough time, be able to dump and read the decrypted key.

7.4 Users

Finally there is a threat of attackers that do not want to extract our key but just want to use our function E. If an attacker is also the original user he also has his password. It seems impossible to protect against these users. The only thing an attacker would have to do is run E and while the function is inside our NDK class it is still possible to run the NDK function from the Java side of the app.

Chapter 8

Conclusions

While our algorithm and introduction of the NDK makes it a lot harder to retrieve our secret key, it still is very much possible for someone with root access to correctly time a memory dump and read the decrypted secret. Because of limitations the key is still in plaintext in memory while the cryptographic operations are begin performed. This does however require specialised knowledge that lots of attacker do not have. If we look at the added code complexity we see that it does require some work to add a NDK side to an app but for an experienced Android developer it should not be that hard, especially when using existing crypto libraries. Another thing worth taking into consideration is that re-factoring an already existing project would take much more time. When looking at a fairly complex project such as the IRMA app it may not be worth the time and added complexity over the added security. In that case, and any case where this is possible, looking at server-side storage and login in with asymmetric cryptography would be a better spending of that time.

Bibliography

[1] Android sharedpreference security - stack overflow. http://stackoverflow.com/questions/9244318/android-sharedpreference-security. (Accessed on 05/21/2016).

[2] cryptography - recommended # of iterations when using pkbdf2-sha256? - information security stack exchange. http://security.stackexchange.com/questions/3959/recommended-of-iterations-when-using-pkbdf2-sha256. (Accessed on 06/02/2016).

[3] Obfuscating android applications using o-llvm and the ndk. http://fuzion24.github.io/android/obfuscation/ndk/llvm/o-llvm/2014/07/27/android-obfuscation-o-llvm-ndk/. (Accessed on 06/06/2016).

[4] Pbkdf2 - wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/PBKDF2. (Accessed on 04/11/2016).

[5] pci dss - which is the best cipher mode and padding mode for aes encryption? - information security stack exchange. http://security.stackexchange.com/questions/52665/which-is-the-best-cipher-mode-and-padding-mode-for-aes-encryption. (Accessed on 05/09/2016).

[6] Dorian Cussen. Android security: The forgetful keystore – systemdotrun – dorian cussen's super blog. https://doridori.github.io/android-security-the%20forgetful-keystore/, february 2015. (Accessed on 04/06/2016).

[7] Simon Judge. Use the android ndk for security sensitive code — android security.guru. http://www.androidsecurity.guru/use-the-android-ndk-for-security-sensitive-code/. (Accessed on 03/24/2016).

[8] Michael Ramirez. Hiding secrets in android apps. https://rammic.github.io/2015/07/28/hiding-secrets-in-android-apps/. (Accessed on 06/01/2016).

[9] Rene Ritchie. iphone vs. android and hardware encryption — imore. http://www.imore.com/iphone-vs-android-and-hardware-encryption. (Accessed on 06/02/2016).

[10] Joeri de Ruiter Time Cooijmans and Erik Poll. Analysis of secure key storage solutions on android. *Security and Privacy in Smartphones and Mobile Devices (SPSM'2014)*, pages 11–20, 2014.