THIS IS JUST AN EXAMPLE OF RESEARCH

Android & iOS storage options

Security

iOS

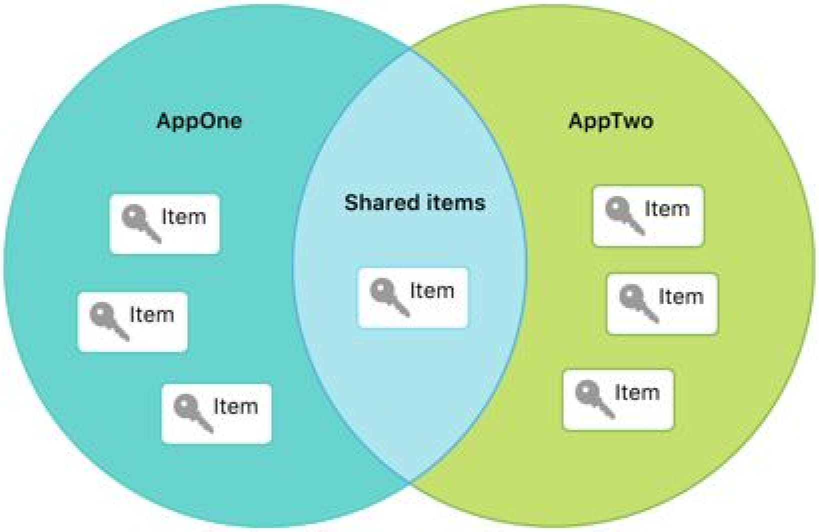
Have to make a own version of secure storage since “keychain access groups” is not yet supported in xamarin.essentials. [8. April 2019] https://github.com/xamarin/Essentials/issues/712. Other issuse (with not so official persons) says that it uses keychain access groups no matter what. https://github.com/AzureAD/azure-activedirectory-library-for-dotnet/issues/1395

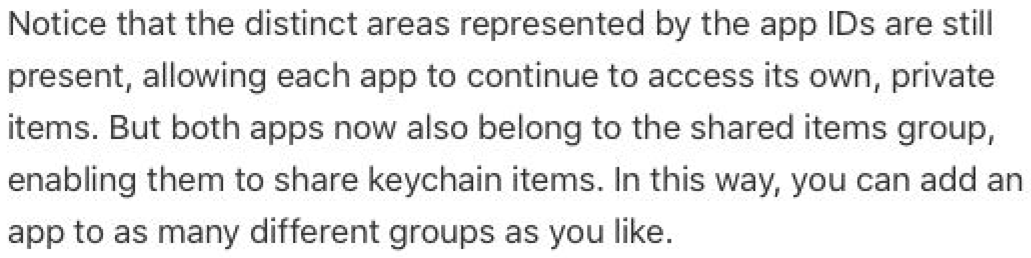
Using Secure Storage is Linker safe, which means it only uses the part of Xamarin essentials we will be using. The rest will be linked away. https://www.eekay.nl/index.php/2019/implement-secure-storage-xamarin-forms-mobile-app/?doing_wp_cron=1554465872.9655671119689941406250

[UPDATE, invalidates almost everything about app groups and keychain below] App groups and keychain access groups

https://developer.apple.com/documentation/security/keychain_services/keychain_items/sharing_access_to_keychain_items_among_a_collection_of_apps

Keychain groups enables an group of apps to share keychain items, while still having private keychain items. An app can be apart of as many keychain groups as it wants, but any keychain item is only part of one keychain group. Fra linken over:





Notice that the distinct areas represented by the app IDs are still present, allowing each app to continue to access its own, private items. But both apps now also belong to the shared items group, enabling them to share keychain items. In this way, you can add an app to as many different groups as you like.

App groups lets you share more, including keychain items. It is often used to share data between an app and an extension. This sharing is distinct from any sharing done with keychain groups. If you use app groups already, it's no need for keychain access groups. You might use keychain access groups to limit sharing between apps.

“[...] order matters. The system considers the first item in the list of access groups to be the app's default access group. This is the access group that keychain services assumes if you don't otherwise specify one when adding keychain items. An app group can't ever be the default, because the app ID is always present and appears earlier in the list. However, a keychain access group can be the default, because it appears before the app ID. In particular, the first keychain access group, if any, that you specify in the corresponding capability becomes the app's default access group. If you don't specify any keychain access groups, then the app ID is the default.” “If you don't specify any access group, keychain services applies your app's default access group, which is the first group named in the concatenated list of groups described in Set Your App's Access Groups.”

It kind of looks like to different statements at the end?

Conclusion

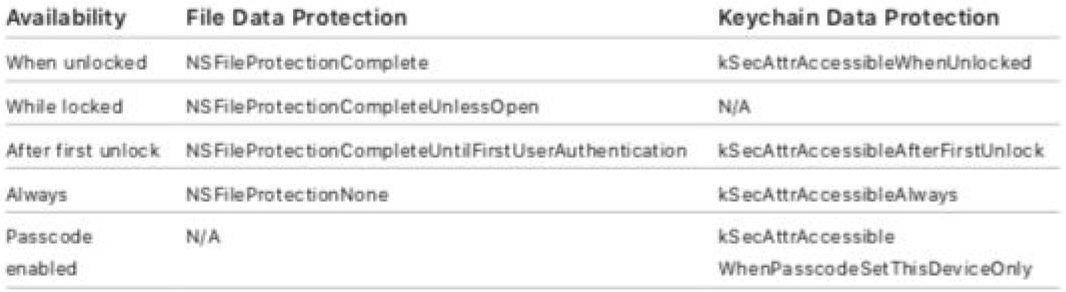
Looks like keychain access groups is perfect for us.

Is it secure to NOT write pin code when switching apps?

The pin code is the most secure way to protect a phone. Adding Face ID or Touch ID does not make it more secure. “Touch ID and Face ID don’t replace your passcode, but provide easy access to your device within thoughtful boundaries and time constraints.“[1]

We know from Runar that the phones in production will lock automatically as fast as possible when not being used. This is set to a 30 seconds timer. This way we can be very sure that no evil person steals the phone and use it to retrieve a/the access token.

All files are assigned a data protection class (which is inside the app sandbox. See security architecture diagram). We will use keychain as it is optimized to store small amounts of text. It has protection classes similar to the protection classes for files. “These classes have behaviors equivalent to file Data Protection classes, but use distinct keys and are part of APIs that are named differently.”



Availability	File Data Protection	Keychain Data Protection
When unlocked	NSFileProtectionComplete	kSecAttrAccessibleWhenUnlocked
While locked	NSFileProtectionCompleteUnlessOpen	N/A
After first unlock	NSFileProtectionCompleteUntilFirstUserAuthentication	kSecAttrAccessibleAfterFirstUnlock
Always	NSFileProtectionNone	kSecAttrAccessibleAlways
Passcode enabled	N/A	kSecAttrAccessible WhenPasscodeSetThisDeviceOnly

“Apps that utilize background refresh services can usekSecAttrAccessibleAfterFirstUnlock for Keychain items that need to be accessed during background updates.” This is the default for third party apps. This protects from boot up attacks

Data protections class: “Complete Protection

(NSFileProtectionComplete): The class key is protected with a key derived from the user passcode and the device UID. Shortly after the
user locks a device (10 seconds, if the Require Password setting is Immediately), the decrypted class key is discarded, rendering all data in this class inaccessible until the user enters the passcode again or unlocks the device using Touch ID or Face ID.“

“The class kSecAttrAccessibleWhenPasscodeSetThisDeviceOnlybehaves the same as kSecAttrAccessibleWhenUnlocked; however, it is available only when the device is configured with a passcode. This class exists only in the system keybag; they:
- Don’t sync to iCloud Keychain
- Aren’t backed up

---

[1] https://www.apple.com/business/site/docs/iOS_Security_Guide.pdf

• Aren’t included in escrow keybags.”

“For example, a VPN certificate must always be available so the device keeps a continuous connection, but it’s classified as “non-migratory,” so it can’t be moved to another device.” VPN passwords has the AfterFirstUnlock class. Meanwhile the iTunes backup has the When unlocked, non-migratory class. Soscial media accont tokens is stored with the AfterFirstUnlock class, which somewhat similar to what we will be storing.

Storing the token in the keychain will keep it there after a restart of the device.

Since we (almost always?) will fetch a new access token after starting the device the difference between AfterFirstUnlock and Always doesn't seem that big.

So the security decisions we have to make is if the access token is needed in the background when the phone is locked. If not, we should use the kSecAttrAccessible WhenPasscodeSetThisDeviceOnly class or the kSecAttrAccessibleWhenUnlocked class. The first one forces the user to have an passcode. The device will have that regardless. The other difference is if the data is synced to iCloud and the MDM (via the escrow keybag). This makes it so the access token can be remoteliy deleted by the MDM.

“The Keychain is implemented as a SQLite database stored on the file system. There is only one database and the *securityd* daemon determines which Keychain items each process or apps can access. Keychain access APIs result in calls to the daemon, which queries the app’s “Keychain- access-groups,” “application-identifier,” and “application-group” entitlements. Rather than limiting access to a single process, access groups allow Keychain items to be shared between apps.”[2]

“Access to items can also be limited by specifying that Touch ID or Face ID enrollment hasn’t changed since the item was added. This limitation helps prevent an attacker from adding their own fingerprint in order to access a Keychain item. “

Conclusion

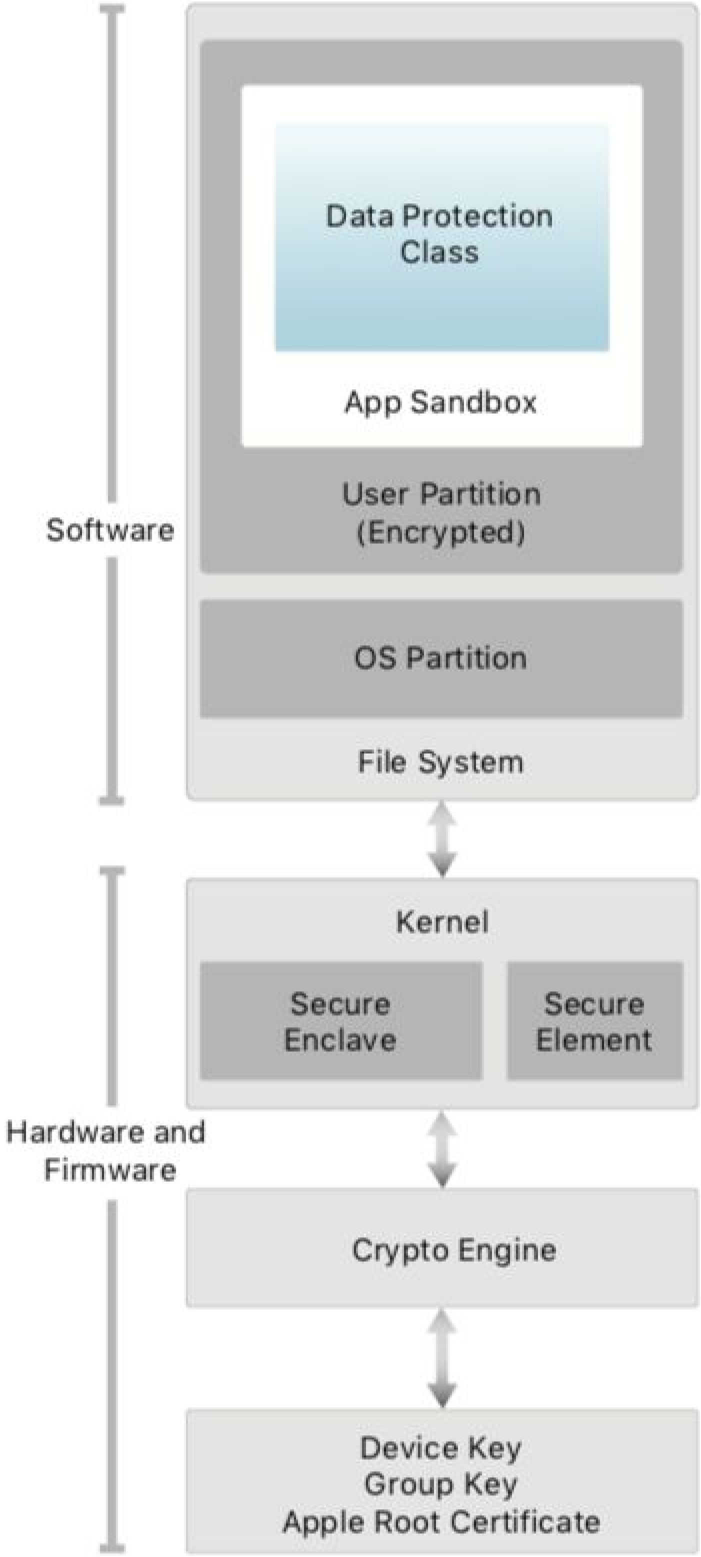
We think that as long as the device has been unlocked from a secure state, the user is authorized. The authorized user does not need to re-authorize between apps. This is because the short timer before the phone locks. We should use the WhenUnlocked class because we (probably) don’t need to use token in background when device is locked. Also for the functionality with mdm to remotely delete token on device (with token invalidation on the server this does not provide any extra security).

Apple official security guide: https://www.apple.com/business/site/docs/iOS_Security_Guide.pdf

Oversikt over sikkerhetsarkitekturen på iOS:

---

[2] https://www.apple.com/business/site/docs/iOS_Security_Guide.pdf


Data Protection Class
App Sandbox
User Partition (Encrypted)
Software
OS Partition
File System
Kernel
Secure Enclave
Secure Element
Hardware and Firmware
Crypto Engine
Device Key
Group Key
Apple Root Certificate

Under System security:

Pointer Authentication Codes

Pointer authentication codes (PACs) are used to protect against exploitation of memory corruption bugs. System software and built-in apps use PAC to prevent modification of function pointers and return addresses (code pointers). Doing so increases the difficulty of many attacks. For example, a Return Oriented Programming (ROP) attack attempts to trick the device into executing existing code maliciously by manipulating function return addresses stored on the stack.
PAC is supported on A12 and S4 SoCs.

Se Architecture overview. Står om fillagring der. Alle filene er kryptert hver for seg.
Se også data protection classes

Ser ut som keychain er den beste løsningen. Bare viktig å bestemme seg for riktig sikkerhetsklasse. Se hva apple selv bruker til sosiale medier tokens. Bare apper fra samme utvikler kan dele data på keychain.

Se også App security - app code signing om bedrifter

Paper on Mobile Device Management (MDM) solutions for providing extra OS security, and the counter-attacks.
https://www.blackhat.com/docs/us-16/materials/us-16-Tan-Bad-For-Enterprise-Attacking-BYOD-Enterprise-Mobile-Security-Solutions-wp.pdf

Paper on adding Secure Storage on top of Android: http://www.ijetch.org/vol8/880-ST011.pdf

App Groups

Apps and extensions owned by a given developer account can share content when configured to be part of an App Group. It is up to the developer to create the appropriate groups on the Apple Developer Portal and include the desired set of apps and extensions. Once configured to be part of an App Group, apps have access to the following:

- A shared on-volume container for storage, which stays on the device as long as at least one app from the group is installed
- Shared preferences
- Shared Keychain items

The Apple Developer Portal guarantees that App Group IDs are unique across the app ecosystem.[3]

---

[3] "iOS Security iOS 12.1 November 2018 - Apple." https://www.apple.com/business/site/docs/iOS_Security_Guide.pdf. Accessed 26 Mar. 2019.

KeyChain security

All of built-in data protection mechanisms on the iPhone, iPad, and iPod touch require the user to set a passcode.

That point is so important it is worth restating for emphasis: without a passcode, all data on the device — including sensitive data stored in the Keychain — can be read by anyone with momentary physical access to the device.

Starting with iOS 8, Apple provided a new kSecAttrAccessible value that allows apps to store items in the Keychain only when the device has a passcode: kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly. Additionally, if the user removes her passcode, all the Keychain items with this access restriction will be removed as well, preventing the items from being exposed on the unprotected device.

Commercial forensic solutions, such as Cellebrite UFED Touch, provide “passcode recovery” tools that will brute-force 4-digit PINs in 20-30 minutes on a standard PC. When a device is protected by a complex, alpha-numeric password, the tools provide an easy-to-use interface for performing a dictionary attack. If the passcode is not resistant to such attacks, then the attacker will be able to de-crypt the Keychain and gain access to the app’s passwords and keys stored within it.

(To enable complex passwords on your device, open the Settings app, select the Passcode or Touch ID & Passcode menu item, and disable the Simple Passcode option.)
Unfortunately, there is no Apple-sanctioned way for an app to determine the complexity of its device’s passcode. However, devices under control of a mobile device manager (“MDM”) can have configuration profiles installed that require passcodes to meet certain rules, such as minimum length.

It is worth noting that Touch ID does not change the strength of the device’s passcode — it only makes it easier to enter. A device using a 4-digit PIN as its passcode is not made more secure by enabling Touch ID. However, the presence of Touch ID encourages use of stronger passcodes, since it removes most of the inconvenience related to entering those passcodes.[4]

KeyChain access restrictions

Starting with iOS 4, each item in the Keychain can have its own access restriction defining when the item can be accessed — the item is securely encrypted the rest of the time (assuming a

---

[4] "iOS Security: Protecting the iOS Keychain - SPR." 16 Apr. 2015, https://spr.com/ios-security-protecting-the-ios-keychain/. Accessed 26 Mar. 2019.

strong passcode has been set for the device). The options are detailed below, from least-restrictive to most-restrictive. The first three options have variants ending with ThisDeviceOnly that prevent the Keychain item from being included in iTunes backups, iCloud backups, or iCloud Keychain. The ThisDeviceOnly variant is commonly used by corporate apps that must protect corporate data, while the standard values are commonly used by apps available through the App Store to give the user control of her data and enable easy device migration.[5]
For more on that read here:.

KeyChain security on jailbroken devices (Pre A7-chip/Secure Enclave)

The Keychain's contents are not secure on a jailbroken device. When a device is jailbroken, the sandbox that limits each app to its own Keychain items can be circumvented, so any code executed on the device while the device is unlocked can access all Keychain items for all apps.

For some apps, it may be sufficient to detect if the device has been jailbroken whenever the app is launched. If the device is jailbroken, the app can respond appropriately; for example, by deleting all its Keychain items and data files. Prateek Gianchandani and Trustwave have good introductions to detecting if an app is executing on a jailbroken device.

Unfortunately, no jailbreak detection method is completely reliable. Additionally, the detection routine can only run when the app is launched or receives background calls to its app delegate. If the user jailbreaks her device after the app has stored sensitive information in the Keychain, then the information will be vulnerable until the next time the app runs.

If it is unacceptable for the app's sensitive data to be exposed this way, then the app must roll its own security and force the user to provide an app-specific password when needed. Implementing such a solution is beyond the scope of this article, but Chapter 10 ("Implementing Encryption") in Jonathan Zdziarski's Hacking and Securing iOS Applications provides a solid introduction to the topic.[6]

Regarding backups of the KeyChain

In reality, all the data backed up is always there, whether you encrypt the backup or not. The only difference is how it is encrypted. In password-protected backups, encryption is based on the user-provided backup password. In backups without password, a hardware key is being used, which is unique for every device. That key is very difficult to obtain. You can only extract

---

[5] "iOS Security: Protecting the iOS Keychain - SPR." 16 Apr. 2015, https://spr.com/ios-security-protecting-the-ios-keychain/. Accessed 26 Mar. 2019.
[6] "iOS Security: Protecting the iOS Keychain - SPR." 16 Apr. 2015, https://spr.com/ios-security-protecting-the-ios-keychain/. Accessed 26 Mar. 2019.

that key using physical acquisition, and even that only works for 32-bit devices, up to and including iPhone 5 and 5c.[7]

KeyChain Security on jailbroken devices (After A7 chip/Secure Enclave)

Devices running iOS 7 or iOS 8 on the Apple A7 and later A-series processors leverage Apple's new "Secure Enclave" technology. The Secure Enclave is a coprocessor that performs security-sensitive tasks — such as verifying the user's passcode and encrypting/decrypting keychain content — without interference from malicious programs.

In practical terms, devices with a Secure Enclave cannot be jailbroken without the user's consent; i.e., while the device is locked or powered off. This significantly impedes attackers attempting to gain access to sensitive data by jailbreaking such a device. However, it does not prevent a user willingly jailbreaking her own device.

This makes the Keychain much more secure in environments that have external consequences to discourage users from jailbreaking, such as company policies for corporate-owned and "BYOD" devices.

Additionally, the Secure Enclave enforces a 5-second delay between failed attempts to unlock the device. This provides a governor against brute-force attacks in addition to safeguards enforced by iOS.[8]

From iOS 12.1 documentation:[9]

The secure enclave does all encryption related operations. From 12.1 doc: "The Secure Enclave provides all cryptographic operations for Data Protection key management and maintains the integrity of Data Protection even if the kernel has been compromised."

Newest version is now 12.2, so some of this information may be outdated. Most likely mostly will not.

---

[7] "iOS 11: jailbreaking, backups, keychain, iCloud – what's the deal ...." 14 Sep. 2017, https://blog.elcomsoft.com/2017/09/ios-11-jailbreaking-backups-keychain-icloud-whats-the-deal/. Accessed 26 Mar. 2019.

[8] "iOS Security: Protecting the iOS Keychain - SPR." 16 Apr. 2015, https://spr.com/ios-security-protecting-the-ios-keychain/. Accessed 26 Mar. 2019.

[9] "iOS Security iOS 12.1 November 2018 - Apple." https://www.apple.com/business/site/docs/iOS_Security_Guide.pdf. Accessed 26 Mar. 2019.

With the exception of the Apple A8 and earlier SoCs, each Secure Enclave generates its own UID (Unique ID) during the manufacturing process. Because the UID is unique to each device and because it’s generated wholly within the Secure Enclave instead of in a manufacturing system outside of the device, the UID isn’t available for access or storage by Apple or any of its suppliers.

For example, the key hierarchy protecting the file system includes the UID, so if the memory chips are physically moved from one device to another, the files are inaccessible. The UID isn’t related to any other identifier on the device.

All wrapped file key handling occurs in the Secure Enclave; the file key is never directly exposed to the application processor. By setting up a device passcode, the user automatically enables Data Protection. The passcode is entangled with the device’s UID, so brute-force attempts must be performed on the device under attack. Due to a time delay it would take more than five and a half years to try all combinations of a six-character alphanumeric.

To improve security while maintaining usability, iOS 11.4.1 or later requires Touch ID, Face ID, or passcode entry to activate the USB interface if USB hasn’t been used recently. This eliminates attack surface against physically connected devices such as malicious chargers while still enabling usage of USB accessories within reasonable time constraints. If more than an hour has passed since the iOS device has locked or since a USB connection has been detached, the device won’t allow any new connections to be established until the device is unlocked.

In addition, on iOS 12 if it’s been more than three days since a USB connection has been established, the device will disallow new USB connections immediately after it locks.

Data protection classes

There are several, but this is my favorite:
Complete Protection (NSFileProtectionComplete): The class key is protected with a key derived from the user passcode and the device UID. Shortly after the user locks a device (10 seconds, if the Require Password setting is Immediately), the decrypted class key is discarded, rendering all data in this class inaccessible until the user enters the passcode again or unlocks the device using Touch ID or Face ID.[10]

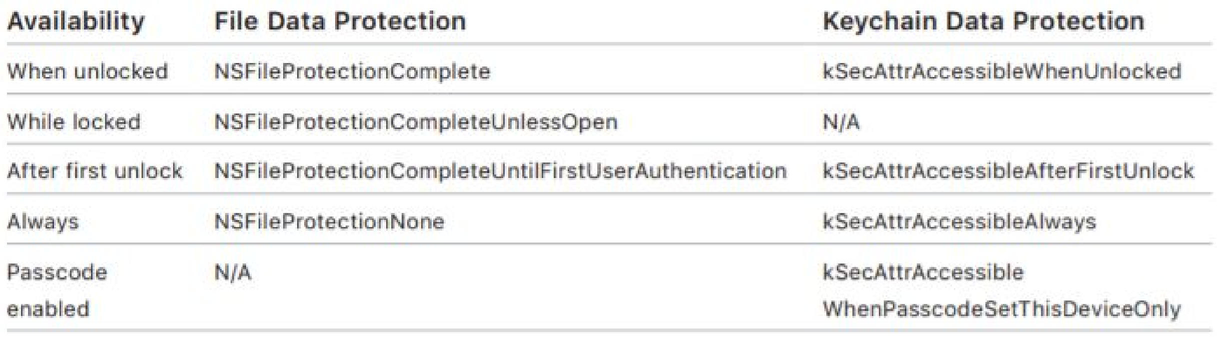
KeyChain

Rather than limiting access to a single process, access groups allow Keychain items to be shared between apps. Keychain items can only be shared between apps from the same developer.

---

[10] "iOS Security iOS 12.1 November 2018 - Apple." https://www.apple.com/business/site/docs/iOS_Security_Guide.pdf. Accessed 26 Mar. 2019.

Keychain data is protected using a class structure similar to the one used in file Data Protection. These classes have behaviors equivalent to file Data Protection classes, but use distinct keys and are part of APIs that are named differently.



Availability	File Data Protection	Keychain Data Protection
When unlocked	NSFileProtectionComplete	kSecAttrAccessibleWhenUnlocked
While locked	NSFileProtectionCompleteUnlessOpen	N/A
After first unlock	NSFileProtectionCompleteUntilFirstUserAuthentication	kSecAttrAccessibleAfterFirstUnlock
Always	NSFileProtectionNone	kSecAttrAccessibleAlways
Passcode enabled	N/A	kSecAttrAccessible WhenPasscodeSetThisDeviceOnly

Apps that utilize background refresh services can use kSecAttrAccessibleAfterFirstUnlock for Keychain items that need to be accessed during background updates.

Access to items can also be limited by specifying that Touch ID or Face ID enrollment hasn't changed since the item was added. This limitation helps prevent an attacker from adding their own fingerprint in order to access a Keychain item. [11]

Data Protection classes

When a new file is created on an iOS device, it's assigned a class by the app that creates it. Each class uses different policies to determine when the data is accessible. The basic classes and policies are described in the following sections.

Complete Protection

(NSFileProtectionComplete): The class key is protected with a key derived from the user passcode and the device UID. Shortly after the
user locks a device (10 seconds, if the Require Password setting is Immediately), the decrypted class key is discarded, rendering all data in this class inaccessible until the user enters the passcode again or unlocks the device using Touch ID or Face ID.

Protected Unless Open

(NSFileProtectionCompleteUnlessOpen): Some files may need to
be written while the device is locked. A good example of this is a mail attachment downloading in the background. This behavior is achieved by using asymmetric elliptic curve cryptography (ECDH over Curve25519). The usual per-file key is protected by a key derived using One-Pass Diffie-Hellman Key Agreement as described in NIST SP 800-56A.

The ephemeral public key for the agreement is stored alongside the wrapped per-file key. The KDF is Concatenation Key Derivation Function (Approved Alternative 1) as described in 5.8.1 of NIST SP 800-56A. AlgorithmID is omitted. PartyUInfo and PartyVInfo are the ephemeral and static public keys, respectively. SHA-256 is used as the hashing function. As soon as the file is closed, the per-file key is wiped from memory. To open the file again, the shared secret is re-created using the Protected Unless Open class's private key and the file's ephemeral public key, which are used to unwrap the per-file key that is then used to decrypt the file.
iOS Security | November 2018 19

---

[11] "iOS Security iOS 12.1 November 2018 - Apple." https://www.apple.com/business/site/docs/iOS_Security_Guide.pdf. Accessed 26 Mar. 2019.

Protected Until First User Authentication

(NSFileProtectionCompleteUntilFirstUserAuthentication):
This class behaves in the same way as Complete Protection, except that the decrypted class key isn't removed from memory when the device
is locked. The protection in this class has similar properties to desktop full-volume encryption, and protects data from attacks that involve a reboot. This is the default class for all third-party app data not otherwise assigned to a Data Protection class.

No Protection

(NSFileProtectionNone): This class key is protected only with the UID, and is kept in Effaceable Storage. Since all the keys needed to decrypt files in this class are stored on the device, the encryption only affords the benefit of fast remote wipe. If a file isn't assigned a Data Protection class, it is still stored in encrypted form (as is all data on an iOS device).

Data protection class key

Class A Complete Protection	(NSFileProtectionComplete)
Class B Protected Unless Open	(NSFileProtectionCompleteUnlessOpen)
Class C ProtectedUntilFirstUserAuthentication	NSFileProtectionCompleteUntilFirstUserAuthentication)
Class D No Protection	(NSFileProtectionNone)

Secure enclave

Does all the
https://www.idownloadblog.com/2017/08/18/apple-wont-fix-iphone-5s-secure-enclave-decryption-key/

Getting KeyChain via iCloud

The thing that was earlier considered impossible (acquiring the keychain for a 64-bit device with a strong iTunes backup password) is in your hands now. Yes, iOS 11 is now supported. Here is the step by step manual to get into the iCloud keychain:[12]
https://blog.elcomsoft.com/2017/08/how-to-extract-icloud-keychain-with-elcomsoft-phone-breaker/

The original Apple ID and password are required. Obviously, a one-time security code is also required in order to pass Two-Factor Authentication, if enabled.[13]

---

[12] "iOS 11: jailbreaking, backups, keychain, iCloud – what's the deal ...." 14 Sep. 2017, https://blog.elcomsoft.com/2017/09/ios-11-jailbreaking-backups-keychain-icloud-whats-the-deal/. Accessed 26 Mar. 2019.

[13] "iOS 11: jailbreaking, backups, keychain, iCloud – what's the deal ...." 14 Sep. 2017, https://blog.elcomsoft.com/2017/09/ios-11-jailbreaking-backups-keychain-icloud-whats-the-deal/. Accessed 26 Mar. 2019.

KeyChain Security in general

For apps that must protect third-party or corporate data, the iOS Keychain can provide adequate security when all of the following conditions are met:

- the sensitive items are stored with an access restriction that makes them only available when the device is unlocked
- the device has a strong passcode
- the device has an Apple A7 and later A-series processor
- there exists a real-world monitoring and response mechanism that discourages the user from jailbreaking the device[14]

There is a big difference between 32-bit and 64-bit devices. For 32-bit devices (up to iPhone 5/5C), we can extract everything including ALL records in the keychain, and even including records that are not available (i.e. cannot be decrypted) in iTuned backups.

For 64-bit devices (starting from iPhone 5S and up to iPhone 7 and iPhone 7 Plus), however, physical acquisition is limited to copying the file system only. The keychain (or some system components that are required to access it) are well protected with Secure Enclave.

As a result, if strong password is set on the iTunes backup and the given device cannot be jailbroken (or even if it can, but it is an iPhone 5S or later), you have no chance to access the keychain.[15]

Android

Source:
https://academy.realm.io/posts/secure-storage-in-android-san-francisco-android-meetup-2017-najafzadeh/
https://developer.android.com/guide/topics/data/data-storage

How fingerprint authentication is secure:
https://android-developers.googleblog.com/2015/10/new-in-android-samples-authenticating.html

---

[14] "iOS Security: Protecting the iOS Keychain - SPR." 16 Apr. 2015, https://spr.com/ios-security-protecting-the-ios-keychain/. Accessed 26 Mar. 2019.

[15] "iOS 11: jailbreaking, backups, keychain, iCloud – what's the deal ...." 14 Sep. 2017, https://blog.elcomsoft.com/2017/09/ios-11-jailbreaking-backups-keychain-icloud-whats-the-deal/. Accessed 26 Mar. 2019.

Scary scenarios

Android rooting gives you superuser access. You then have to ability to change app privileges and potentially access other app's processes with native methods.

Two main scenarios

Attacker takes a unlocked phone. With an unlocked phone it is easier to gain root access. There are two main alternatives: many android phones can be rooted with an app. There is no unchangeable way to restrict app downloading from other sources than the Play Store. Or, the attacker can enable USB debugging and connect the device to a computer.

Attacker takes a locked phone. Might be harder to root the device if the attacker can't gain access to developer options to enable USB debugging or download a malicious app. **CHECK IF BOOT VERIFIER CAN MITIGATE THIS.**

Possible attacks

If user authentication is not required for using private keys stored in either SE or TEE an attacker could add an malicious app and, with root access, somehow set its UID to be the same as the genuine app. Therefore being able to pose as the genuine app and use its keys.

Condition for this to work:

1. Attacker must have acquired the encrypted access token from somewhere.

If user authentication is required then an attacker could attempt to trace the memory space of the running app with a native function like ptrace(requires root access)[16]. This might enable the attacker to see the access token, from a memory dump, when it's in the genuine app's running process.

Condition for this to work:

1. Attacker must be able to root the device without deleting any data from the phone or at least be able to restore it to its former state, so it appears unchanged to the user.

4 storage options:

- Internal file storage
    - Store app-private files on the device file system.

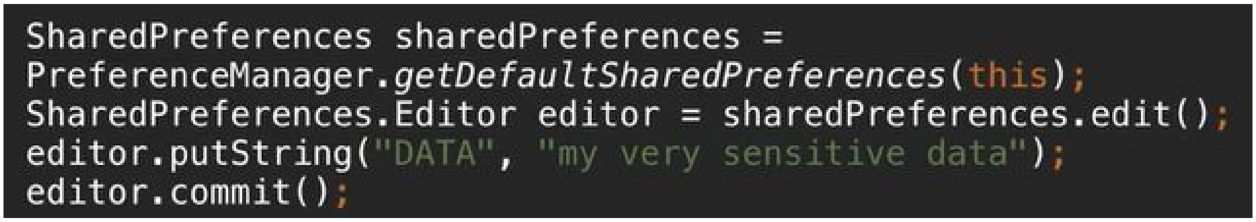
---

[16] "ptrace(2) - Linux manual page - man7.org." http://man7.org/linux/man-pages/man2/ptrace.2.html. Accessed 26 Mar. 2019.

- External file storage
    - Store files on the shared external file system. Easily share files with other apps.
- Shared preferences
    - Store private primitive data in key-value pairs.
- Databases
    - Store structured data in a private database.

Except for external file storage, all of these are intended for storing app-private data. Use a Content Provider to share files from these private locations with other apps.

Shared Preferences



```
SharedPreferences sharedPreferences =
PreferenceManager.getDefaultSharedPreferences(this);
SharedPreferences.Editor editor = sharedPreferences.edit();
editor.putString("DATA", "my very sensitive data");
editor.commit();
```

Info:

- If you don't need to store a lot of data and it doesn't require structure, you should use this. If you have a relatively small collection of key-values that you'd like to save, you should use the SharedPreferences APIs
- Uses XML files that persist across user sessions, even if the app is killed.
- NOTE: Name is misleading, SharedPreferences are used to save ANY simple data.
- Can be private or shared.

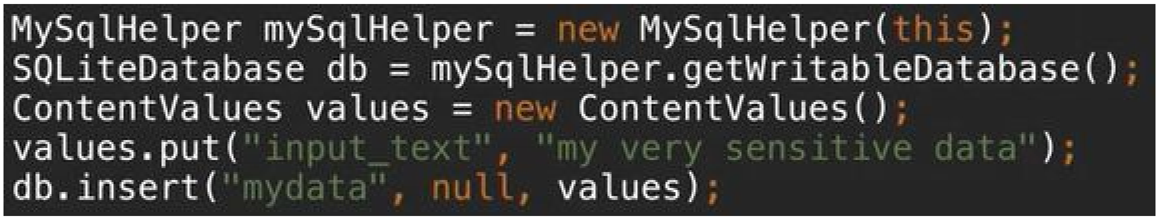
Pros:

- Easy to use
- “Private”: Data is not exposed to other applications

Conclusion:

Probably the best fit for our need.

Databases(SQLite)



```
MySqlHelper mySqlHelper = new MySqlHelper(this);
SQLiteDatabase db = mySqlHelper.getWritableDatabase();
ContentValues values = new ContentValues();
values.put("input_text", "my very sensitive data");
db.insert("mydata", null, values);
```

A lot more is done before this.

Info:

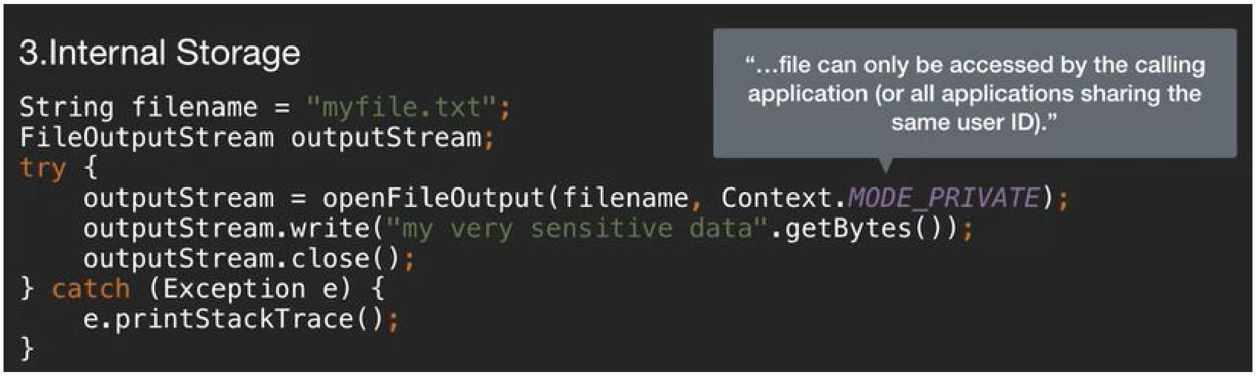
- Accessible only by your app by default.
- Intended for saving a large amount of structured data.

Pros:

Cons:
- Data can be grabbed by rooting the phone.

Conclusion:
Not intended for storing a small amount of strings. Probably not useful for us.

Internal Storage



3.Internal Storage

```
String filename = "myfile.txt";
FileOutputStream outputStream;
try {
    outputStream = openFileOutput(filename, Context.MODE_PRIVATE);
    outputStream.write("my very sensitive data".getBytes());
    outputStream.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

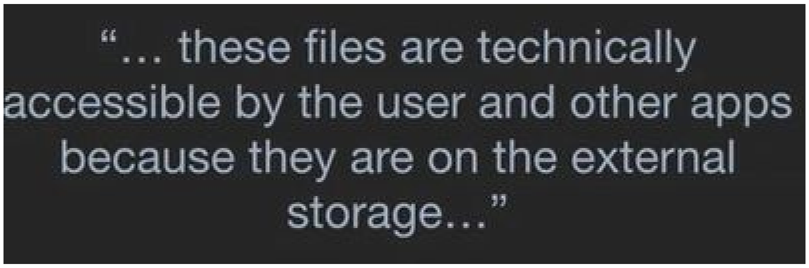
"…file can only be accessed by the calling application (or all applications sharing the same user ID)."

Info:
- Data can be read on rooted device.
- Files saved here are only accessible from the app that saved them. User can’t access these from anywhere else. Although in earlier API levels you can set them to be accessible to other apps.
- Saved in a private directory.
- Files saved here are automatically removed if app is uninstalled.

Internal cache files:
- Can be used to store data temporarily, although they might be deleted by the OS.

Conclusion:
Might be good for our use.

External Storage



"… these files are technically accessible by the user and other apps because they are on the external storage…"

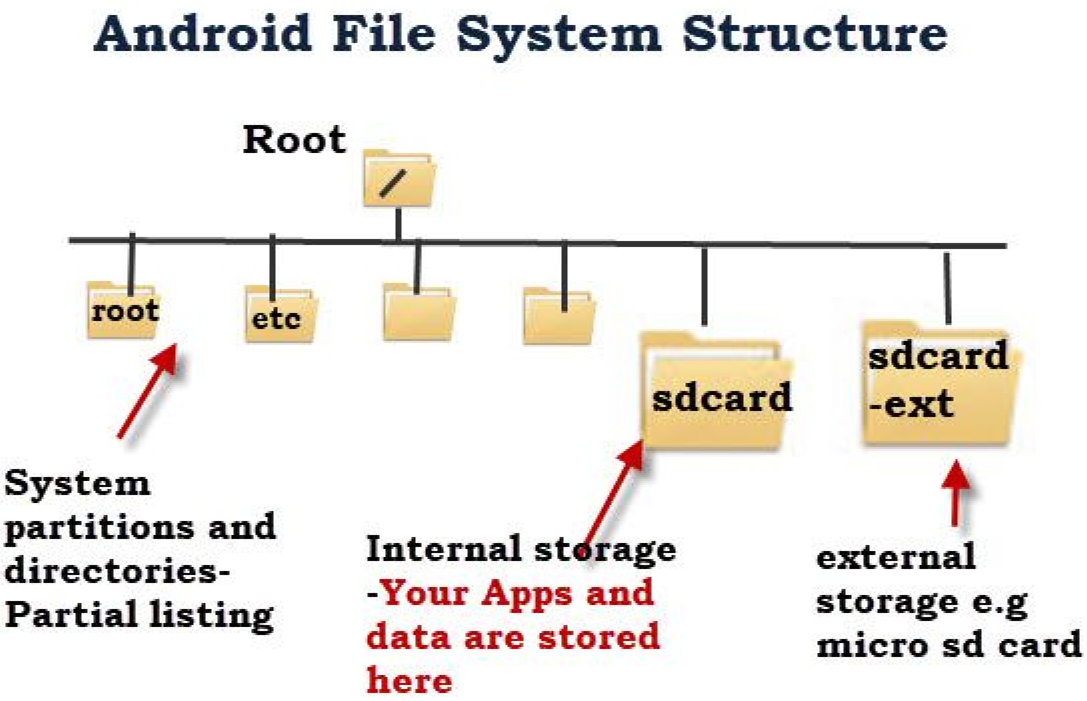
Info:

- Files stored here can be read by anyone.
- Most often, you should use external storage for user data that should be accessible to other apps and saved even if the user uninstalls your app, such as captured photos or downloaded files.

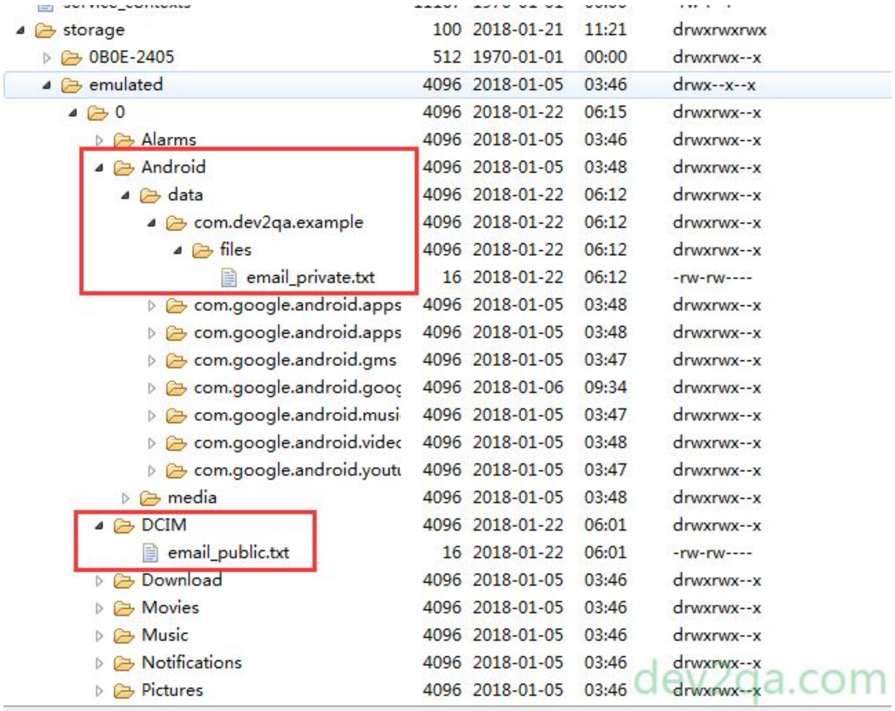
Conclusion:

Not intended for storing sensitive, temporary and app-private data. Probably shouldn't use this.

Implementing external storage


Android File System Structure
Root
/
root
etc
sdcard
sdcard -ext
System partitions and directories- Partial listing
Internal storage -Your Apps and data are stored here
external storage e.g micro sd card

Android external storage can store files in SD card or device build-in flash disk memory. Internal and external is only a logical classification.[17]



```
◢ storage                         100  2018-01-21  11:21  drwxrwxrwx
  ▷ 0B0E-2405                     512  1970-01-01  00:00  drwxrwx--x
  ◢ emulated                     4096  2018-01-05  03:46  drwx--x--x
    ◢ 0                          4096  2018-01-22  06:15  drwxrwx--x
      ▷ Alarms                   4096  2018-01-05  03:46  drwxrwx--x
      ◢ Android                  4096  2018-01-05  03:48  drwxrwx--x
        ◢ data                   4096  2018-01-22  06:12  drwxrwx--x
          ◢ com.dev2qa.example   4096  2018-01-22  06:12  drwxrwx--x
            ◢ files              4096  2018-01-22  06:12  drwxrwx--x
                email_private.txt  16  2018-01-22  06:12  -rw-rw----
          ▷ com.google.android.apps  4096  2018-01-05  03:48  drwxrwx--x
          ▷ com.google.android.apps  4096  2018-01-05  03:48  drwxrwx--x
          ▷ com.google.android.gms   4096  2018-01-05  03:47  drwxrwx--x
          ▷ com.google.android.goo   4096  2018-01-06  09:34  drwxrwx--x
          ▷ com.google.android.musi  4096  2018-01-05  03:47  drwxrwx--x
          ▷ com.google.android.vide  4096  2018-01-05  03:48  drwxrwx--x
          ▷ com.google.android.yout  4096  2018-01-05  03:47  drwxrwx--x
        ▷ media                  4096  2018-01-05  03:48  drwxrwx--x
      ◢ DCIM                     4096  2018-01-22  06:01  drwxrwx--x
          email_public.txt         16  2018-01-22  06:01  -rw-rw----
      ▷ Download                 4096  2018-01-05  03:46  drwxrwx--x
      ▷ Movies                   4096  2018-01-05  03:46  drwxrwx--x
      ▷ Music                    4096  2018-01-05  03:46  drwxrwx--x
      ▷ Notifications            4096  2018-01-05  03:46  drwxrwx--x
      ▷ Pictures                 4096  2018-01-05  03:46  drwxrwx--x
```

“Using the external storage is great for files that you want to share with other apps or allow the user to access with a computer.

After you request storage permissions and verify that storage is available, you can save two different types of files:

- Public files: Files that should be freely available to other apps and to the user. When the user uninstalls your app, these files should remain available to the user. For example, photos captured by your app or other downloaded files should be saved as public files.
- Private files: Files that rightfully belong to your app and will be deleted when the user uninstalls your app. Although these files are technically accessible by the user and other apps because they are on the external storage, they don't provide value to the user outside of your app.

Caution: The external storage might become unavailable if the user removes the SD card or connects the device to a computer. And the files are still visible to the user and other apps that have the **READ_EXTERNAL_STORAGE** permission. So if your app's functionality depends on

---

[17] "Android Read Write External Storage File Example." 22 Jan. 2018, https://www.dev2qa.com/android-read-write-external-storage-file-example/. Accessed 28 Mar. 2019.

these files or you need to completely restrict access, you should instead write your files to the internal storage."[18]

Saving to a public repository

If you want to save public files on the external storage, use the getExternalStoragePublicDirectory() method to get a File representing the appropriate directory on the external storage. The method takes an argument specifying the type of file you want to save so that they can be logically organized with other public files, such as DIRECTORY_MUSIC or DIRECTORY_PICTURES.

If you want to hide your files from the Media Scanner, include an empty file named `.nomedia` in your external files directory (note the dot prefix in the filename). This prevents media scanner from reading your media files and providing them to other apps through the MediaStore content provider.

Creating custom repository in external storage

When you create Android applications that require to use external storage of the phone to store files, you will need to create folders in the phone' external storage. In Android, you can get the path of the external storage by using the getExternalStorageDirectory() method of the Environment class. Then you can use the mkdir() or mkdirs() method of the File class to create a folder in the external storage. If the parent folder of the folder to be created already exists, you will use the mkdir() method. Otherwise, you must use the mkdirs() method. The mkdir() method will only create the bottom most folder. Your folder won't be created if its parent folder doesn't exist in the directory structure. You can use the exists() method of the File class to check whether the folder to be created already exists before calling the mkdir() or mkdirs() method. It returns true if the folder already exists. Otherwise it returns false. The mkdir() or mkdirs() method returns true if the folder is successfully created. Otherwise, it returns false.[19]

```
String folder_main = "NewFolder";

File f = new File(Environment.getExternalStorageDirectory(), folder_main);
if (!f.exists()) {
   f.mkdirs();
}
```
[20]

---

[18] "Save files on device storage | Android Developers." https://developer.android.com/training/data-storage/files. Accessed 28 Mar. 2019.

[19] "How to create a folder in internal and external storage ...." 24 Feb. 2018, https://www.quora.com/How-do-I-create-a-folder-in-internal-and-external-storage-programmatically-in-an-Android-app. Accessed 28 Mar. 2019.

[20] "how to create a folder in android External Storage Directory ...." https://stackoverflow.com/questions/24781213/how-to-create-a-folder-in-android-external-storage-directory. Accessed 28 Mar. 2019.

Data sharing between apps

Sharing simple data with intents

Specify an intent with ACTION_SEND to indicate that you want to send data from one app to another, even across process boundaries. You can send the Uri pointing to the data you want the other app to retrieve/use. To do this the receiving app needs to have permission to access it. To ways to do this:

- Store the data in your own ContentProvider, making sure the other apps have the correct permissions to access your ContentProvider. Use the FileProvider class to easily create a ContentProvider with this functionality.
- Use the system MediaStore. Content here is available to all apps so not suited for our use.

Sharing data with startActivityForResult

Once the sub-activity finishes, the onActivityResult() method in the calling activity is called.

Sharing files

In all cases, the only secure way to offer a file from your app to another app is to send the receiving app the file's content URI and grant temporary access permissions to that URI. Content URIs with temporary URI access permissions are secure because they apply only to the app that receives the URI, and they expire automatically.[21]

FileProvider

When using a FileProvider the user of our library must add a <provider> element to the AndroidManifest.xml and possibly a xml-file with the paths(defined by us maybe?) to the stored tokens.

ContentProvider

Possible uses-permissions is set in the manifest file of the application with the ContentProvider. The app with the ContentResolver(app asking for content) will declare needed permissions in their manifest, which the user will implicitly grant when installing the app.

---

[21] "Sharing files | Android Developers." https://developer.android.com/training/secure-file-sharing. Accessed 21 Mar. 2019.

Sharing token with Chrome Custom Tab cookie

It’s not possible to set the cookie from a client app.[22] The Custom Tabs use the Chrome browser cookie store which is not possible to set from a client app.

Permissions

Permissions can be defined by the <permission> element contained in the <manifest> element. You can set the required permission to use a ContentProvider or Activity with the <permission> element contained in either the <provider> or <activity> element. An app wanting to use either the ContentProvider or Activity then have to set the <uses-permission> element contained in <manifest> to be the same as the previously defined permission.
The permission level can be set to three different values:

Signature

A permission that the system grants only if the requesting application is signed with the same certificate as the application that declared the permission. If the certificates match, the system automatically grants the permission without notifying the user or asking for the user's explicit approval.[23] Can be useful to make our own certificate to sign all the DIPS apps. Automatically grants permissions to send data between them.

Dangerous

Requires user approval during runtime.

Normal

Automatically granted during runtime.

SignatureOrSystem

A permission that the system grants only to applications that are in a dedicated folder on the Android system image *or* that are signed with the same certificate as the application that declared the permission. Avoid using this option, as the signature protection level should be sufficient for most needs and works regardless of exactly where apps are installed[24]

---

[22] "539240 - Being able to set/get cookies for a domain that client app ...." https://bugs.chromium.org/p/chromium/issues/detail?id=539240. Accessed 27 Mar. 2019.

[23] "<permission> | Android Developers." https://developer.android.com/guide/topics/manifest/permission-element. Accessed 22 Mar. 2019.

[24] "<permission> | Android Developers." https://developer.android.com/guide/topics/manifest/permission-element. Accessed 28 Mar. 2019.

Sharing everything between 2+ apps (Same android:sharedUserId)

android:sharedUserId in <manifest>
The name of a Linux user ID that will be shared with other apps. By default, Android assigns each app its own unique user ID. However, if this attribute is set to the same value for two or more apps, they will all share the same ID — provided that their certificate sets are identical. Apps with the same user ID can access each other's data and, if desired, run in the same process.

Android:process in <application>
“By setting this attribute to a process name that's shared with another application, you can arrange for components of both applications to run in the same process — but only if the two applications also share a user ID and be signed with the same certificate.”

“Note that in order to retain security, only two apps signed with the same signature (and requesting the same sharedUserId) will be given the same user ID.[25] Any data stored by an app will be assigned that app's user ID, and not normally accessible to other packages.”

http://java-hamster.blogspot.com/2010/05/androids-shareduserid.html (2010)

Android Keystore System

Tutorial with demo (outdated w. API 28):
https://medium.com/overmorrow/authentication-sucks-bad-security-too-345ed20463d4

Keystore features

The keystore can generate private keys that never leave the separate secure hardware. The data to be decrypted is moved to the secure hardware and decrypted in the memory there. Impossible to extract the key. Although, if you somehow manage to pose as the legitimate app you can ask it to decrypt the access token for you. This might be possible if an attacker manages to gain superuser privileges. With this you may be able to gain access to the app’s process, even though it runs in its own sandbox on a kernel-level.

The Android KeyStore allows an application to store cryptographic keys that can’t be extracted from the application process or the Android device as a whole.[26]

---

[25] "Define a Custom App Permission | Android Developers." https://developer.android.com/guide/topics/permissions/defining. Accessed 28 Mar. 2019.

[26] "Android keystore system | Android Developers." https://developer.android.com/training/articles/keystore. Accessed 27 Mar. 2019.

The Keystore itself is encrypted using the user's own lockscreen pin/password, hence, when the device screen is locked the Keystore is unavailable. Keep this in mind if you have a background service that could need to access your application secrets.[27]

Keys generated by the KeyMaster (Keymaster Hardware Abstraction Layer (HAL)) are encrypted using a hardware-backed encryption key and returned to the OS.

"If the app's process is compromised, the attacker may be able to use the app's keys but cannot extract their key material (for example, to be used outside of the Android device)."[28] The only secure way to store encrypted sensitive data would then be to always require some kind of unfakeable authentication every time the the app access the keystore. Is the AlwaysRequireAuthentication method unchangeable after generating the key? Is it possible to change the biometric data of the authenticated user? Key use authorizations might mitigate this; authorizations are set when the key is generated and are unchangeable. [29]

"Use the Android Keystore provider to let an individual app store its own credentials that only the app itself can access. This provides a way for apps to manage credentials that are usable only by itself while providing the same security benefits that the KeyChain API provides for system-wide credentials. This method requires no user interaction to select the credentials."[30]

It seems the private key generated and stored in the keystore is only ever accessible from the app that created it. If we are to be able to send the encrypted access token from any app we want we have to be able to decrypt it from any app. The solution to this might be to retrieve the access token from the first app, where it can be decrypted, for every HTTP request.

Version Binding

Version binding ensures that a generated key are only usable to a device running the exact same OS version it was generated with.

KeyStore API:
https://developer.android.com/reference/android/security/keystore/package-summary

---

[27] "How to use the Android Keystore to store passwords and other ...." 10 Jul. 2015, https://www.androidauthority.com/use-android-keystore-store-passwords-sensitive-information-623779/. Accessed 25 Mar. 2019.

[28] "Android keystore system | Android Developers." https://developer.android.com/training/articles/keystore. Accessed 25 Mar. 2019. Under extraction prevention

[29] "Android keystore system | Android Developers." https://developer.android.com/training/articles/keystore. Accessed 25 Mar. 2019. Under key use authorizations

[30] "Android keystore system | Android Developers." https://developer.android.com/training/articles/keystore. Accessed 22 Mar. 2019.

Does the user need to enter a pin code every time the app is used? (Or when switching apps)

When generating or importing a key into the AndroidKeyStore you can specify that the key is only authorized to be used if the user has been authenticated. The user is authenticated using a subset of their secure lock screen credentials (pattern/PIN/password, fingerprint).

When a key is authorized to be used only if the user has been authenticated, it is configured to operate in one of the two modes:

1. Timed validity. Keys become authorized for use as soon as the user unlocks the device. “When the user has a secure lockscreen enabled and pulls out his phone in order to check his account balance, he can just walk right into your app without being penetrated by a second authentication screen inside your app. That’s because he just authenticated on system level when he unlocked his phone.”[31]
   Validity can be set on a per-key basis and have a validity timeout before re-authentication is required. These keys become permanently invalidated once the secure lock screen is disabled (reconfigured to None, Swipe or other mode which doesn't authenticate the user) or forcibly reset (e.g. by a Device Administrator).

   a. Use: **setUserAuthenticationValidityDurationSeconds() (API 23)** to enter this mode
      Requires:
      - `setUserAuthenticationRequired(boolean)`
      - Handling UserNotAuthenticatedException by using `KeyguardManager#createConfirmDeviceCredentialIntent(CharSequence, CharSequence)`
        - WARNING: Deprecated in API Level Q
        - Use **`BiometricPrompt.Builder#setEnableFallback(boolean)`** (API level Q) with Activity.startActivityForResult(Intent, int) checking for Activity.RESULT_OK. The user will first be prompted to authenticate with biometrics, but also given the option to authenticate with their device PIN, pattern, or password
        - See: https://developer.android.com/reference/android/hardware/biometrics/BiometricPrompt.Builder.html

2. Per-use-authentication. Keys cannot be used for cryptographic operations unless the user authorizes it. Currently, the only means of such authorization is fingerprint

---

[31] "Authentication sucks. Bad security too. – overmorrow – Medium." 30 Oct. 2016, https://medium.com/overmorrow/authentication-sucks-bad-security-too-345ed20463d4. Accessed 31 Mar. 2019.

authentication. Such keys can only be generated or imported if at least one fingerprint is enrolled. These keys become permanently invalidated once a new fingerprint is enrolled or all fingerprints are unenrolled.[32]
By setting KeyGenParameterSpec.Builder.setUserAuthenticationRequired to true, you can permit the use of the key only after the user authenticates with the user's fingerprint. [33]

Use: **setUserAuthenticationRequired() (API 23)**[34]
Requires:

- `KeyguardManager#isDeviceSecure()`,
- `setUserAuthenticationValidityDurationSeconds(int)`,
- `FingerprintManager#hasEnrolledFingerprints()`.
- Wrapping the call in a FingerprintManager.CryptoObject, invoking FingerprintManager.authenticate.[35]
- WARNING: DEPRECATED
- **FingerprintManager was deprecated in API level 28.**
- See **`BiometricPrompt`** which shows a system-provided dialog upon starting authentication. In a world where devices may have different types of biometric authentication, it's much more realistic to have a system-provided authentication dialog since the method may vary by vendor/device.
- More on BiometricPrompt: https://www.androidauthority.com/add-fingerprint-authentication-app-biometricprompt-943784/
- Possibly working Xamarin Android code: https://gist.github.com/justintoth/49484bb0c3a0494666442f3e4ea014c0

Other factors

Key can be protected with the following methods when they are being created:

setUnlockedDeviceRequired (Android 9.0)
Sets whether the keystore requires the screen to be unlocked before allowing decryption using this key. If this is set to true, any attempt to decrypt or sign using this key while the screen is locked will fail. A locked device requires a PIN, password, fingerprint, or other trusted factor to

---

[32] "Android keystore system | Android Developers." https://developer.android.com/training/articles/keystore. Accessed 31 Mar. 2019.
[33] "GitHub - googlesamples/android-AsymmetricFingerprintDialog." https://github.com/googlesamples/android-AsymmetricFingerprintDialog. Accessed 31 Mar. 2019.
[34] "KeyGenParameterSpec.Builder | Android Developers." https://developer.android.com/reference/android/security/keystore/KeyGenParameterSpec.Builder. Accessed 31 Mar. 2019.
[35] "KeyGenParameterSpec.Builder | Android Developers." https://developer.android.com/reference/android/security/keystore/KeyGenParameterSpec.Builder. Accessed 31 Mar. 2019.

access. While the screen is locked, the key can still be used for encryption or signature verification.[36]

setIsStrongBoxBacked (Android 9.0)
Sets whether this key should be protected by a StrongBox security chip.[37]

setUserPresenceRequired (Android 9.0)
Confirmation verifies that some user with physical possession of the device has approved a displayed message. For example, a physical button press can be used as a test of user presence if the other pins connected to the button are not able to simulate a button press. Requires the user to provide input to the secure hardware that cannot be spoofed by the main processor.[38]

setKeyValidityEnd() (API 23)[39]
Sets the time instant after which the key is no longer valid.

Conclusion:

We want the keys unavailable/unusable unless they are being used by an authorized human, for maximum security. **setUnlockedDeviceRequired()** seems to be the most promising alternative since it adds security without requiring any additional steps from the user.

setUserAuthenticationRequired() is also a promising alternative as using the **setUserAuthenticationValidityDurationSeconds()** with it lets us extend the keys usability a few extra seconds to complete login if the user should lock the screen during login. The problem with this is that before API 28, the only method of authentication once the timeout from initially unlocking the device is over, is using fingerprint. While after Android Q, …?? missing sentence ??. There are some devices without a fingerprint sensor.

Both solutions don't explicitly require user authentication when switching apps. But in solution 2, the app will require the user to re-authenticate when the **setUserAuthenticationValidityDurationSeconds()** resource access duration has expired.

---

[36] "KeyGenParameterSpec.Builder | Android Developers." https://developer.android.com/reference/android/security/keystore/KeyGenParameterSpec.Builder. Accessed 31 Mar. 2019.
[37] "KeyGenParameterSpec.Builder | Android Developers." https://developer.android.com/reference/android/security/keystore/KeyGenParameterSpec.Builder. Accessed 31 Mar. 2019.
[38] "KeyGenParameterSpec.Builder | Android Developers." https://developer.android.com/reference/android/security/keystore/KeyGenParameterSpec.Builder. Accessed 31 Mar. 2019.
[39] "KeyGenParameterSpec.Builder | Android Developers." https://developer.android.com/reference/android/security/keystore/KeyGenParameterSpec.Builder. Accessed 31 Mar. 2019.

Hardware security module (StrongBox Keymaster)

Supported devices running Android 9 (API level 28) or higher installed can have a *StrongBox Keymaster*, an implementation of the Keymaster HAL (Hardware Abstraction Layer) that resides in a hardware security module. The module contains the following:

- Its own CPU.
- Secure storage.
- A true random-number generator.
- Additional mechanisms to resist package tampering and unauthorized sideloading of apps.

When checking keys stored in the StrongBox Keymaster, the system corroborates a key's integrity with the Trusted Execution Environment (TEE).

Cryptographic and user authentication authorizations are likely to be enforced by secure hardware. Temporal validity interval authorizations are unlikely to be enforced by the secure hardware because it normally doesn't have an independent secure real-time clock.[40]

Whether a key's user authentication authorization is enforced by the secure hardware can be queried using KeyInfo.isUserAuthenticationRequirementEnforcedBySecureHardware()[41]

KeyChain API

Use the [KeyChain](https://developer.android.com/reference/android/security/KeyChain) API when you want system-wide credentials. When an app requests the use of any credential through the KeyChain API, users get to choose, through a system-provided UI, which of the installed credentials an app can access. This allows several apps to use the same set of credentials with user consent.[42]

Tutorial for using KeyChain: https://nelenkov.blogspot.com/2011/11/using-ics-keychain-api.html

---

[40] "Android keystore system | Android Developers." https://developer.android.com/training/articles/keystore. Accessed 31 Mar. 2019.
[41] "Android keystore system | Android Developers." https://developer.android.com/training/articles/keystore. Accessed 31 Mar. 2019.
[42] "Android keystore system | Android Developers." https://developer.android.com/training/articles/keystore. Accessed 28 Mar. 2019.

Android P (9) enhanced security

Android P introduces the unlockedDeviceRequired flag, which users can set so that the device cannot be decrypted unless the screen is also unlocked.[43]

Attacking Qualcomm KeyStore on rooted device (Android 5.0 to 7.0)

It is possible change the mapping of processes to the keystore.
<UID of the app>_USRPKEY_<key alias given inside app>
<UID of the app>_USRCERT_<key alias given inside app>
By copying the "keyblobs" into another app's directory and changing the UID, the other app can use the keys provided they are still on the same device. Copying the files to another device will not work because the private key can only be decrypted with the hardware-baked key that exists in the TEE.[44]

For Android 5:

- An attacker that has root permissions can easily use the keys of other apps on the same device by renaming the keystore files.
- Key pairs cannot be used outside of the device because the private data are encrypted with a device-specific key living inside the TEE that cannot be exported

For Android 6 and 7:

- It is possible to request user input/consent before the key is to be used.
    - If the attacker knows the user's password, it can access the key.

Attacking software based keymaster (Android 5 to 7)

When a device does not require a PIN to unlock it no encryption is used for the private key. By parsing the USRPKEY file an attacker can learn all information that should be kept secret such as the private exponent and the two primes of the RSA key pair. However, when a PIN is required to unlock the device a random 128-bit AES master key is used for encryption. This master key is randomly generated and stored in the .masterkey file in the directory above. This file is encrypted using a key that is

---

[43] "These Are All Of Android P's New Security Enhancements." 9 May. 2018, https://www.tomshardware.co.uk/android-p-all-security-enhancements,news-58397.html. Accessed 26 Mar. 2019.

[44] "security evaluation of android keystore - Dione - UniPi." http://dione.lib.unipi.gr/xmlui/bitstream/handle/unipi/11252/KASAGIANNIS%20GEORGIOS%201515%20-%20SECURITY%20EVALUATION%20OF%20ANDROID%20KEYSTORE.pdf?sequence=3&isAllowed=y. Accessed 25 Mar. 2019.

derived from the PIN using 8192 rounds of PKCS5 PBKDF2 HMAC SHA1. The master key is used to encrypt all key entries without any form of per-entry keyderivation. So if a password is used to unlock the device even if an attacker does not know the PIN or password, he can brute-force it outside of the device in case it has not much entropy or learn it from memory. Therefore we consider R(device-bound) as violated.[45] (Possibly only Android 5, source unclear)

Android 5, 6 and 7: Plaintext RSA private key recovery possible with root access.

Alternative solution

A safer asymmetric encryption option is to store the private key remotely. In Android this usually means using the Spongy Castle libraries or other alternative such as Google's Keyczar.[46] Important note: Keyczar is deprecated. The Keyczar developers recommend Tink.[47]

What is TEE (trusted execution environment)?

TEE runs its own OS, and communicates with the "main" OS only through a "restricted" interface (generally a shared memory region which both OS's use to exchange data). In the case of Android, this means that compromising the Android OS (or its kernel) would not result in compromising the processes running in the TEE.

Android phones have been equipped with TEEs for some time now, and it is common that the biometric sensors of the phone (e.g. the fingerprint scanner) are directly connected to the TEE, meaning that the end user's biometric data lives only within the TEE and never in the Android OS.[48]

Generated keys are never exposed to the kernel when the device is equipped with a "Trusted Execution Environment". Every new phone running Android Nougat will come with a TEE installed.[49]

---

[45] "security evaluation of android keystore - Dione - UniPi." http://dione.lib.unipi.gr/xmlui/bitstream/handle/unipi/11252/KASAGIANNIS%20GEORGIOS%201515%20-%20SECURITY%20EVALUATION%20OF%20ANDROID%20KEYSTORE.pdf?sequence=3&isAllowed=y. Accessed 25 Mar. 2019.

[46] "Best place to store a password in your Android app - Android Authority." 31 Mar. 2015, https://www.androidauthority.com/where-is-the-best-place-to-store-a-password-in-your-android-app-597197/. Accessed 25 Mar. 2019.

[47] "GitHub - google/keyczar: Easy-to-use crypto toolkit." https://github.com/google/keyczar. Accessed 25 Mar. 2019.

[48] "Android KeyStore: what is the difference between "StrongBox" and ...." 6 Nov. 2018, https://proandroiddev.com/android-keystore-what-is-the-difference-between-strongbox-and-hardware-backed-keys-4c276ea78fd0. Accessed 25 Mar. 2019.

[49] "Authentication sucks. Bad security too. – overmorrow – Medium." 30 Oct. 2016, https://medium.com/overmorrow/authentication-sucks-bad-security-too-345ed20463d4. Accessed 31 Mar. 2019.

AMD TrustZone

With TrustZone, user space applications operate in "normal" mode. The kernel runs "system" mode. The trusted kernel operates in "monitor" mode. Because of this architecture, even a "rooted" application cannot access protected regions within the trusted kernel.

What is a SE (secure element?)

It goes one step further than TEE: it is a separate microchip, with its own CPU, storage, RAM, etc. which is designed specifically for security-relevant purposes. SEs are resistant to a wider variety of attacks, both logical and physical, such as side-channel attacks. Interestingly enough, you are probably using SEs in your daily life without knowing it: your SIM card's chip is a SE, and also is your credit card's.[50]

Is TEE secure enough?

If key info's isInsideSecureHardware() returns true (which would mean that your keys are protected *-at least-* by a TEE), then you can also rest assured that your keys are protected by a "good enough" combination of software and hardware.

TEE does not resist to hardware attacks. It is designed to provide large memory, high processing power, protects its resources against software attacks, even get control on the display and keypad – to guarantee What You See Is What You Sign. But the TEE would not survive attacks from a user opening his mobile phone and questioning it with probed. And to make sure we all agree on the TEE resistance, there is a security certification describing exactly the attacks that a TEE can survive (and security geeks will be happy to know that it corresponds to a EAL2+ common criteria certification.[51]

Unfortunately, there will be lots of devices where neither StrongBox nor hardware-backed keys will be available. What is the solution for those devices? The answer is: it depends. In such devices, the security of the key material relies solely on software measures, which means that a compromise of the Android OS (such as a root exploit) might end up revealing you user's keys.

---

[50] "Android KeyStore: what is the difference between "StrongBox" and ...." 6 Nov. 2018, https://proandroiddev.com/android-keystore-what-is-the-difference-between-strongbox-and-hardware-backed-keys-4c276ea78fd0. Accessed 25 Mar. 2019.

[51] "Trusted Execution Environment, millions of users have one, do you ...." 18 Feb. 2014, https://poulpita.com/2014/02/18/trusted-execution-environment-do-you-have-yours/. Accessed 25 Mar. 2019.

While AndroidKeyStore provides Devicebinding, an attacker can create a signing oracle on the device to query for signatures over arbitrary data. So while the attacker cannot gain access to the private key, he can effectively use it without any limitations. In the end, the additional security that TEE-backed secure key storage provides is Device-binding. It provides security against malicious apps, but not against a root attacker. (2014)

"KeyChain isBoundKeyAlgorithm. Keychain API now provides a method (isBoundKeyType) that allows applications to confirm that system-wide keys are bound to a hardware root of trust for the device. This provides a place to create or store private keys that cannot be exported off the device, even in the event of a root compromise." [52] (2014)

The easiest and cheapest one is to leave the users of such devices behind, disabling the feature that requires the crypto material if the required protection level is not available. In cases were this is not possible, WhiteBox Cryptography (WBC) libraries offer a good compromise between security and device support.[53]

TEE Vulnerabilities

https://duo.com/blog/sixty-percent-of-enterprise-android-phones-affected-by-critical-qsee-vulnerability
https://www.mocana.com/blog/android-keystore-vulnerable-to-compromise
https://threatpost.com/android-keystore-encryption-scheme-broken-researchers-say/119092/

Rooted device with QualComms AndroidKeyStore

Keys can be used on other apps

What is WhiteBox Cryptography?

On open devices, the cryptographic keys used for making a payment are observable and modifiable, rendering them vulnerable to attack. White box cryptography prevents the exposure of confidential information such as these keys. To do so, keys are obfuscated by not only storing them in the form of data and code, but also random data and the composition of the code itself.

Though white box cryptography resists reverse engineering threats to the cryptographic keys, anti-reverse engineering deterrents are also required to ensure the code surrounding the white box cryptography primitives remains intact. This could be done through native machine-code obfuscation, mangling Java Native Interface names and Java byte-code obfuscation. In addition,

---

[52] "Analysis of Secure Key Storage Solutions on Android - Institute for ...." http://www.cs.ru.nl/J.deRuiter/papers/spsm14.pdf. Accessed 25 Mar. 2019.

[53] "Android KeyStore: what is the difference between "StrongBox" and ...." 6 Nov. 2018, https://proandroiddev.com/android-keystore-what-is-the-difference-between-strongbox-and-hardware-backed-keys-4c276ea78fd0. Accessed 25 Mar. 2019.

anti-tamper mechanisms may be applied, such as integrity checking and self-healing. In some cases, these techniques are deployed to detect reverse-engineering tools.[54]

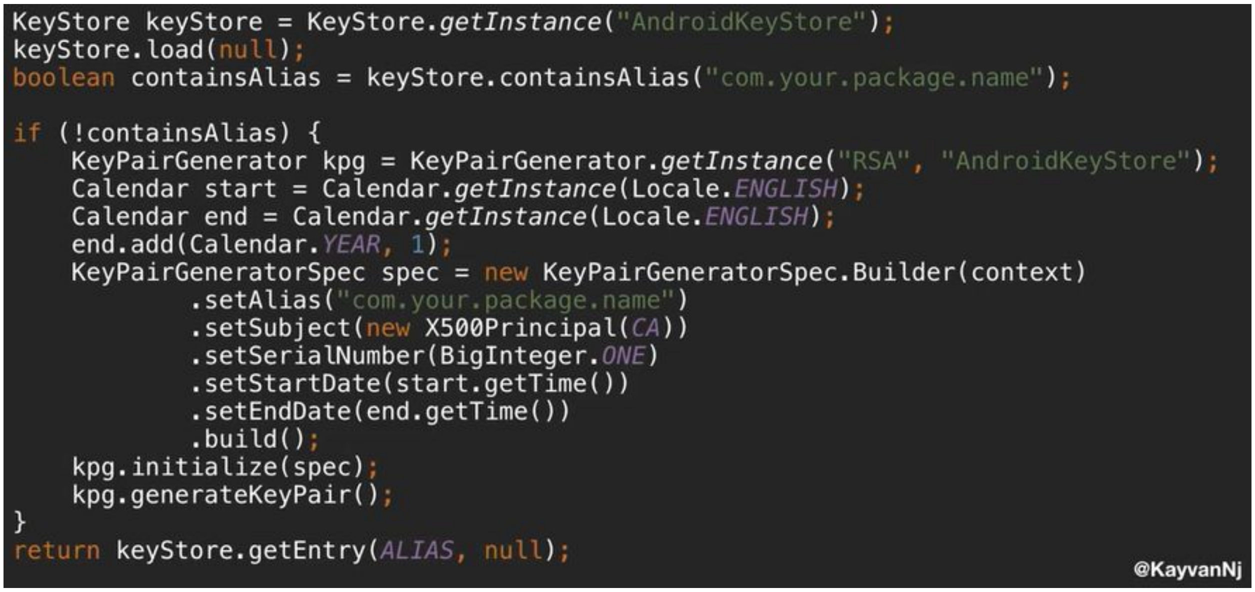
Mitigating attacks on rooted devices

https://www.vantagepoint.sg/blog/75-soft-token-cloning-attacks-and-mitigations

The problem with protecting from root is that root has access to everything your app can access, so even if you encrypt your access token, root would have access to the encryption key and could decrypt it. The same holds when your application is compromised.

One way to add a bit of security might be to have a service which is housed outside the device do an additional layer of encryption and decryption of the access token based on a revocable key, so if the device was lost, you could revoke the associated key and then the encrypted access token on the device would be useless. But for this solution, you'd need to have two layers of encryption, one on the device (otherwise, you'd open up a new vulnerability by sending plaintext access tokens over the network to another service) and one on the service.[55]

How KeyStore works



```
KeyStore keyStore = KeyStore.getInstance("AndroidKeyStore");
keyStore.load(null);
boolean containsAlias = keyStore.containsAlias("com.your.package.name");

if (!containsAlias) {
    KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA", "AndroidKeyStore");
    Calendar start = Calendar.getInstance(Locale.ENGLISH);
    Calendar end = Calendar.getInstance(Locale.ENGLISH);
    end.add(Calendar.YEAR, 1);
    KeyPairGeneratorSpec spec = new KeyPairGeneratorSpec.Builder(context)
            .setAlias("com.your.package.name")
            .setSubject(new X500Principal(CA))
            .setSerialNumber(BigInteger.ONE)
            .setStartDate(start.getTime())
            .setEndDate(end.getTime())
            .build();
    kpg.initialize(spec);
    kpg.generateKeyPair();
}
return keyStore.getEntry(ALIAS, null);
```

@KayvanNj

---

[54] "What is White Box Cryptography - Rambus." 8 Aug. 2018, https://www.rambus.com/blogs/what-is-white-box-cryptography/. Accessed 25 Mar. 2019.

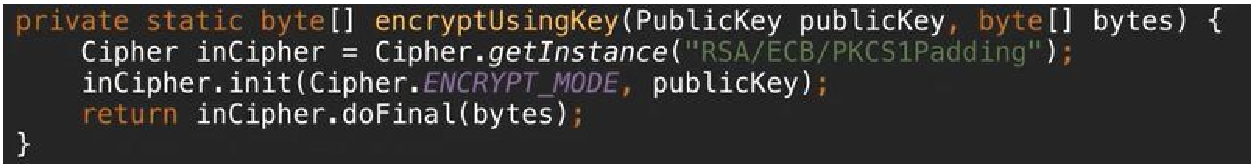
[55] "mobile - How to securely store accesstoken in android ...." 3 Nov. 2016, https://security.stackexchange.com/questions/141696/how-to-securely-store-accesstoken-in-android. Accessed 25 Mar. 2019.

About the code:

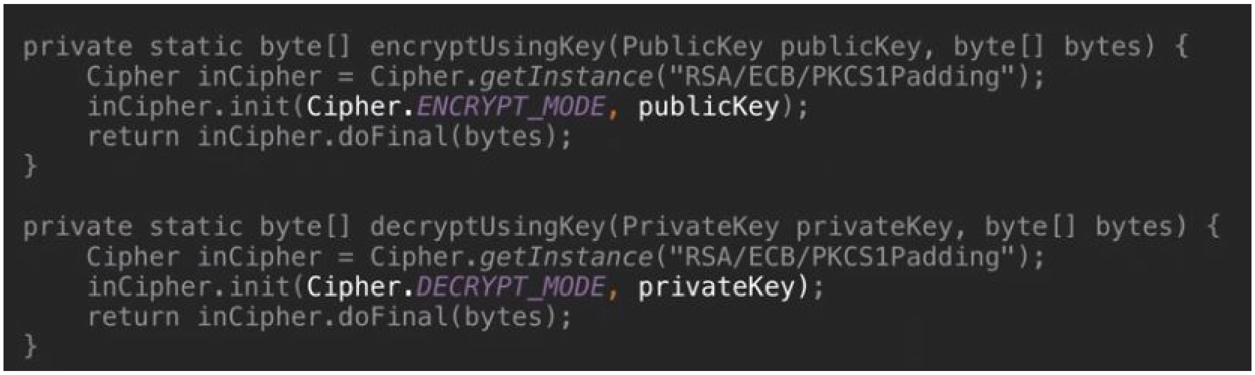
- KeyPairGenerator used to create a private and public key

Info:

- Master key is hardware secured
- Gives us operations to use



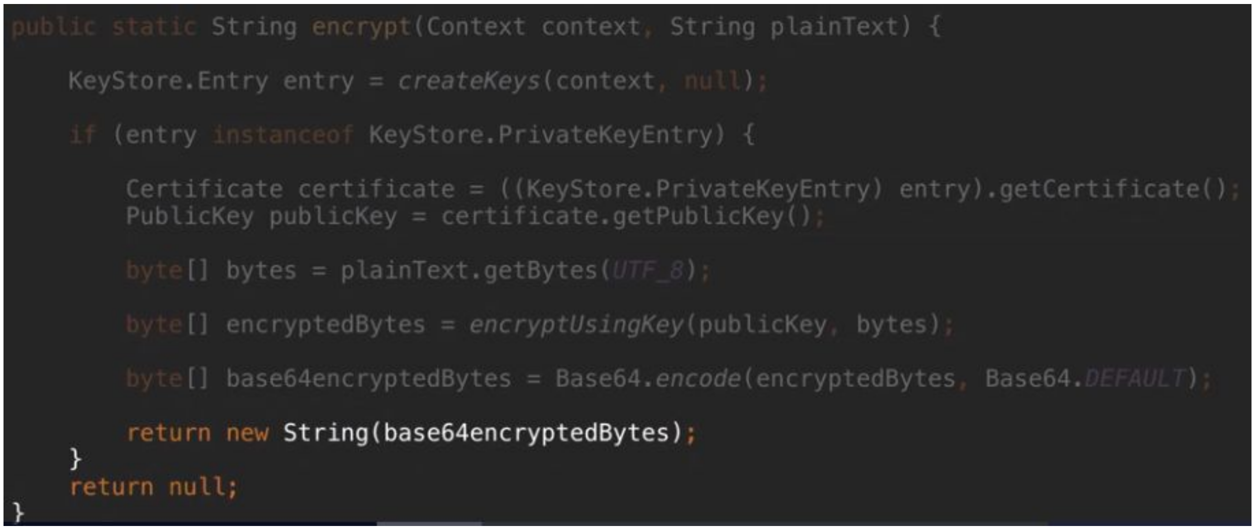
```java
private static byte[] encryptUsingKey(PublicKey publicKey, byte[] bytes) {
    Cipher inCipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
    inCipher.init(Cipher.ENCRYPT_MODE, publicKey);
    return inCipher.doFinal(bytes);
}
```



```java
private static byte[] encryptUsingKey(PublicKey publicKey, byte[] bytes) {
    Cipher inCipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
    inCipher.init(Cipher.ENCRYPT_MODE, publicKey);
    return inCipher.doFinal(bytes);
}

private static byte[] decryptUsingKey(PrivateKey privateKey, byte[] bytes) {
    Cipher inCipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
    inCipher.init(Cipher.DECRYPT_MODE, privateKey);
    return inCipher.doFinal(bytes);
}
```

Only return byte arrays, can be wrapped in:

Encryption:

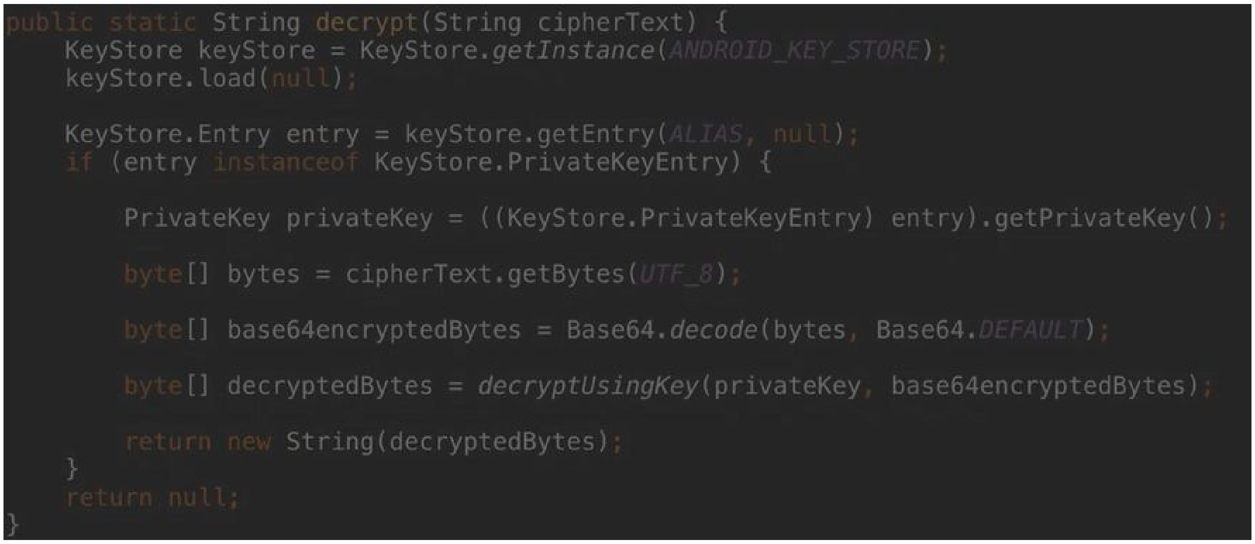


```java
public static String encrypt(Context context, String plainText) {

    KeyStore.Entry entry = createKeys(context, null);

    if (entry instanceof KeyStore.PrivateKeyEntry) {

        Certificate certificate = ((KeyStore.PrivateKeyEntry) entry).getCertificate();
        PublicKey publicKey = certificate.getPublicKey();

        byte[] bytes = plainText.getBytes(UTF_8);

        byte[] encryptedBytes = encryptUsingKey(publicKey, bytes);

        byte[] base64encryptedBytes = Base64.encode(encryptedBytes, Base64.DEFAULT);

        return new String(base64encryptedBytes);
    }
    return null;
}
```

Notes:

- Encode encrypted data with base64 so that it is mapped to ASCII characters

Decryption:



```
public static String decrypt(String cipherText) {
    KeyStore keyStore = KeyStore.getInstance(ANDROID_KEY_STORE);
    keyStore.load(null);

    KeyStore.Entry entry = keyStore.getEntry(ALIAS, null);
    if (entry instanceof KeyStore.PrivateKeyEntry) {

        PrivateKey privateKey = ((KeyStore.PrivateKeyEntry) entry).getPrivateKey();

        byte[] bytes = cipherText.getBytes(UTF_8);

        byte[] base64encryptedBytes = Base64.decode(bytes, Base64.DEFAULT);

        byte[] decryptedBytes = decryptUsingKey(privateKey, base64encryptedBytes);

        return new String(decryptedBytes);
    }
    return null;
}
```

Didn’t finish the video, but this was the most important part.

Notes for this video: https://www.youtube.com/watch?v=jzpCIldtusA

- Token binding so cookies can’t be exfiltrated
    - Token bound cookies
- How it is solved? App links.
    - Can open any part of another app if they are configured for it

Sharing data (03.05.2019)

AccountManager:
https://developer.android.com/reference/android/accounts/AccountManager