Insecure Data Storage in Android Applications

Group 6

Bassam Kaddoura
TODO

Jonathan Persgården
jonpersg@student.chalmers.se

May 28, 2021

Contents

1 Introduction

As more and more parts of society is digitized we use mobile apps to complete an increasing number of tasks using a variety of different apps. Being such a big part of our lives these apps access, use and generate a large amount of data about us. Most of this data might not be of high value, such as user setting etc, but some might be very sensitive. For instance many people have their credit card information stored in at least one of their apps.

While most of this data is usually stored in the cloud, mobile developers usually store some of it locally on the device to reduce loading times, server load and to let the app work as intended even when offline. Furthermore, this avoids potential security hazards by reducing the amount of potential sensitive data sent over the internet.

The local data is often stored in the device non-encrypted or with a key somewhere in the code [1] and could therefore be a target for attackers. In this report we will investigate how Android devices handle stored data and how it can be accessed by an attacker.

2 Motivation

2.1 Problem statement

Many Android applications store very sensitive data on the device [1]. This could cause problems if a person with malicious intents finds or otherwise manages to access someone else's Android device.

2.2 Goal

The goal of this project was to learn how an Android developer can make data stored locally as secure as possible. The work done to accomplish this has been the following:

- Learn how Android applications store local data.
- Use knowledge to learn about possible exploits and their mitigations.
- Explore real-life use cases for found exploits.

3 Background

There are many ways to store data locally in Android [6]. We have chosen to look at two of the most common, Shared Preferences and an SQLite database.

3.1 Shared preferences

When storing key value pairs of relatively low size the recommended method is to use the shard preferences. This lets the developer persist data without much boiler-plate code. Image [2] is an example of how values can be stored in the shared preferences.

These are implemented by XML files stored in plain text in the apps directory within the android device. The file can be set to let other apps access it or to give read-write privileges only to the app that creates it. [11]

3.2 SQLite database

When storing relation data, or structured data in an amount too large for the Shared preferences, the recommended tool is an in-memory SQLite database. It lets the developer create relational tables and interact with them using regular SQL queries.

There are many libraries that developers can use to interact with the database but the currently most common one is the Room library [10] being listed as one of Google's Jetpack libraries[9] provided to make android development demand less boiler-plate code.

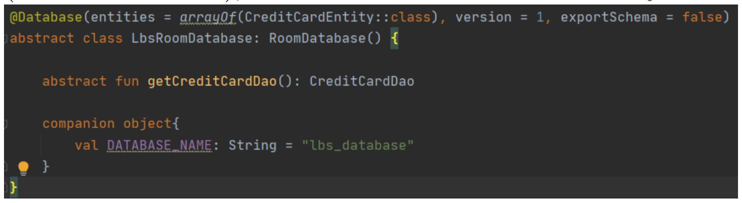
Internally the database is also stored in a file in the apps directory. The database file is only accessible for the app which created it.

4 Prototype

For the purpose of demonstration, an application was built on Kotlin using Android Studio [5], the application was made to utilize both the shared preferences storage, and the local SQLite database. Initially, the application stores the data in plain text, in order to show the flaw in doing that, later on, pieces of code are added that are aimed to solve the vulnerabilities.

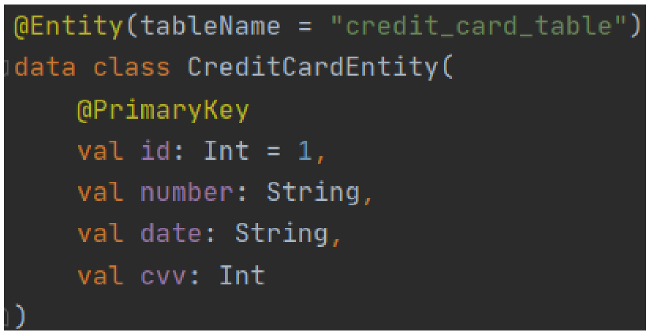
4.1 Database setup

The SQLite database is setup to only store the sensitive data we are targeting (credit card details), so the structure and code are fairly basic:



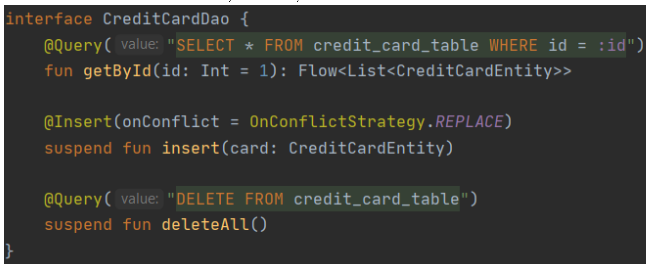
```
@Database(entities = arrayOf(CreditCardEntity::class), version = 1, exportSchema = false)
abstract class LbsRoomDatabase: RoomDatabase() {

    abstract fun getCreditCardDao(): CreditCardDao

    companion object{
        val DATABASE_NAME: String = "lbs_database"
    }
}
```

The above figure shows the creation of the database, the database only stores one entity (credit card), with a function to get a hold of that entity, the database is also saved under the name ”lbs_database”



```kotlin
@Entity(tableName = "credit_card_table")
data class CreditCardEntity(
    @PrimaryKey
    val id: Int = 1,
    val number: String,
    val date: String,
    val cvv: Int
)
```

The above figure shows the creation of the credit card table that will store the credit card number, data, and CVV.

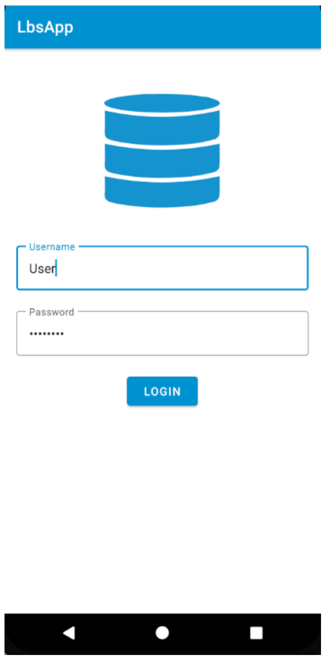


```kotlin
interface CreditCardDao {
    @Query( value: "SELECT * FROM credit_card_table WHERE id = :id")
    fun getById(id: Int = 1): Flow<List<CreditCardEntity>>

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insert(card: CreditCardEntity)

    @Query( value: "DELETE FROM credit_card_table")
    suspend fun deleteAll()
}
```

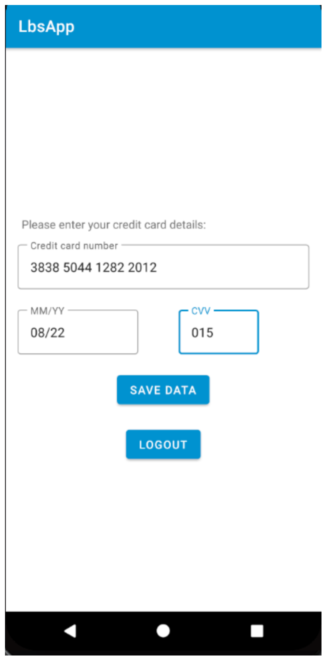
In this figure, the ways we are going to interact with the database are explained, for this demo, we only needed to get, insert, and delete data.

4.2 Frontend

The application is fairly simple, it consists of 2 activities, a login activity, and a main activity. The main activity is where the user will be prompted to enter our target sensitive data, credit card information in this example.



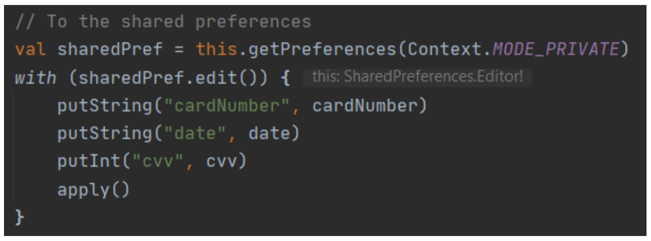
(a) Login Activity



(b) Main Activity

Figure 1: Prototype Preview

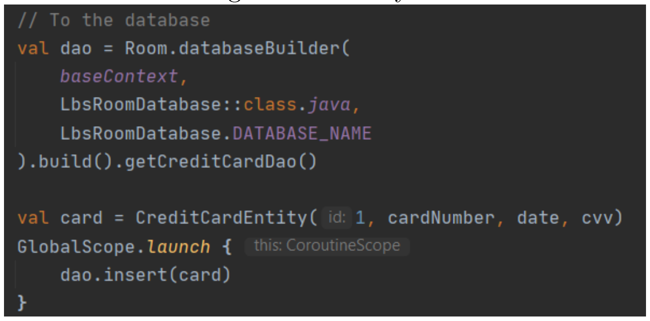
Figure 1a shows the login activity where the user enters their login credentials. Figure 1b shows the main activity where the user inputs the sensitive data, which is represented by credit card information in this prototype, a save button is present to mimic the functionality of remembering credit card information, a function is fired which includes the two pieces of that are responsible for storing the data.



```
// To the shared preferences
val sharedPref = this.getPreferences(Context.MODE_PRIVATE)
with (sharedPref.edit()) {  this: SharedPreferences.Editor!
    putString("cardNumber", cardNumber)
    putString("date", date)
    putInt("cvv", cvv)
    apply()
}
```

Figure 2: Shared Preferences example

This showcases the piece of code responsible for storing the information in the shared preferences storage. An instance of the storage is created, and then each value is stored along with a its key name.



```
// To the database
val dao = Room.databaseBuilder(
    baseContext,
    LbsRoomDatabase::class.java,
    LbsRoomDatabase.DATABASE_NAME
).build().getCreditCardDao()

val card = CreditCardEntity( id: 1, cardNumber, date, cvv)
GlobalScope.launch {  this: CoroutineScope
    dao.insert(card)
}
```

The above code is responsible for storing the information in the local SQLite database. The first 5 lines call for an instance of the credit card table, then a credit card entity is created with the details input from the user, finally that entity is inserted into the table.

If the user chooses to logout, their data will be wiped:
To clear the shared preferences, we use the same code in figure 2, but we replace **putString** with **remove("Key Name")**.
As for the database, an instance of the table is created similar to figure 3, then instead of dao.insert, we use dao.deleteAll to clear the database.

It is also worth noting that an artificial delay of 2000ms is added before the data is wiped for an easier simulation of the attack.

5 Different Attacks and their Mitigations

5.1 Prevent logout from clearing user data

A common feature in Android apps is to let the user log in to access the system using their account. While logged in the user can usually browse or input data that is potentially sensitive. The user expects this data to be visible only while logged in and that the app will clear it once the user has logged out of the app. This attack highlights a case when locally stored data can be accessed even after the user has logged out.

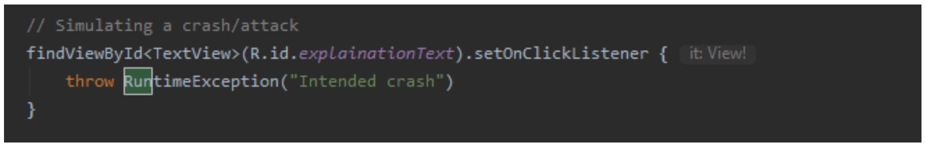
Most android apps will usually only have one user per device since devices are personal. Therefore setting up a data structure where data for multiple users are stored simultaneously is in most cases an unnecessary complication to add to the code-base. Instead, most Android apps clear all data once the user logs out. However, this takes time and if the app would crash during the clearing some data will still be left in storage. Even worse, when the next user logs in the previous user's data might, depending on the app implementation, be displayed. This could be exploited by an attacker by intentionally crash the app during logout.

This vulnerability might not be very severe on its own since it is hard to exploit without having a device that someone else stores sensitive data on and controlling how the previous user log's out. However, it can cause some problems in combination with other vulnerabilities. As a case study to highlight this we can use the GuardTools Mobile app [12] which had both this vulnerability and the one describes in section 5.2. The app is used by security guards and store sensitive data about their work locally on the device. A concerning scenario is that someone steals a device from a guard and attempts to read data from the app. One of the features of the app is that it updates a server with the user's location and actions in the app while the user is logged in. Because of this, it would be hard to steal data from the app while logged in. If the attacker uses this vulnerability to log out and still let the data be stored locally the vulnerability in section 5.2 could be used to silently steal the data.

To the best of our knowledge this attack has not been researched before.

5.1.1 Attack

As mentioned above, to execute this attack one needs to shut down the application during the logout process. This could be done by using a known bug that causes a crash in the application and triggers it during the logout period. Other possible ways are to force shutdown from the device settings or simply by removing the battery. If the app has this vulnerability and if this is executed correctly the app will be in a state where it will be logged out but still store data from the previous user. To exploit this reliably in our prototype app we added a delay on the logout call between where it sets the logged-in state to false and the wiping of the credit card data. Then we added the code below to let us crash the application easily by clicking on some of the text on the screen.



```
// Simulating a crash/attack
findViewById<TextView>(R.id.explainationText).setOnClickListener {  it: View!
    throw RuntimeException("Intended crash")
}
```

When the app is opened up again the previous user's credit card information is indeed loaded from storage and visible to the new user.

5.1.2 Mitigation

There are many possible ways to handle this. One idea is to always wipe sensitive data when the app crashes by adding a default exception handler [7]. This will however affect the user experience since most apps crash from usual bugs relatively often. Furthermore, it will not help if the attacker performs the attack by truing off the device. Another solution that also might cause problems is to wipe all user data and then set the logged-in state to false. This would keep the user safe from the attack but might cause other bugs in the application since it

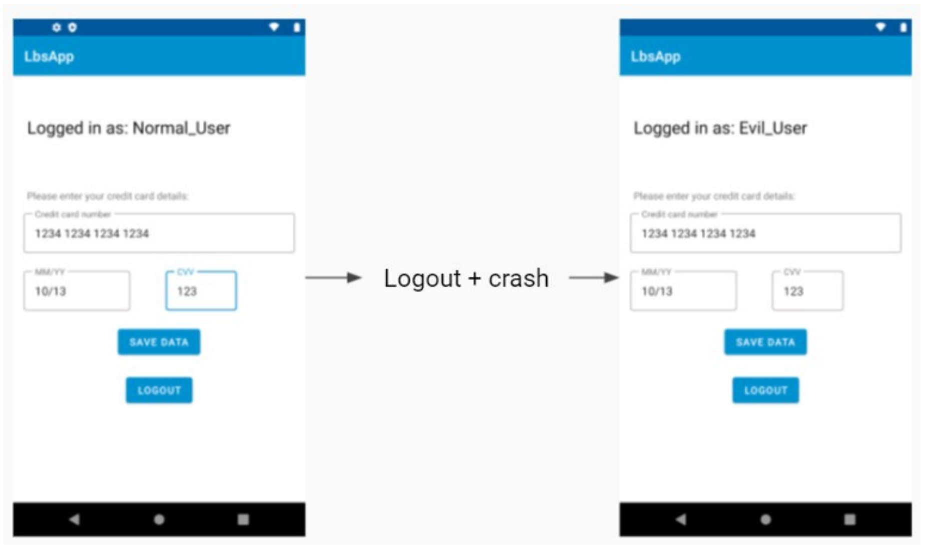
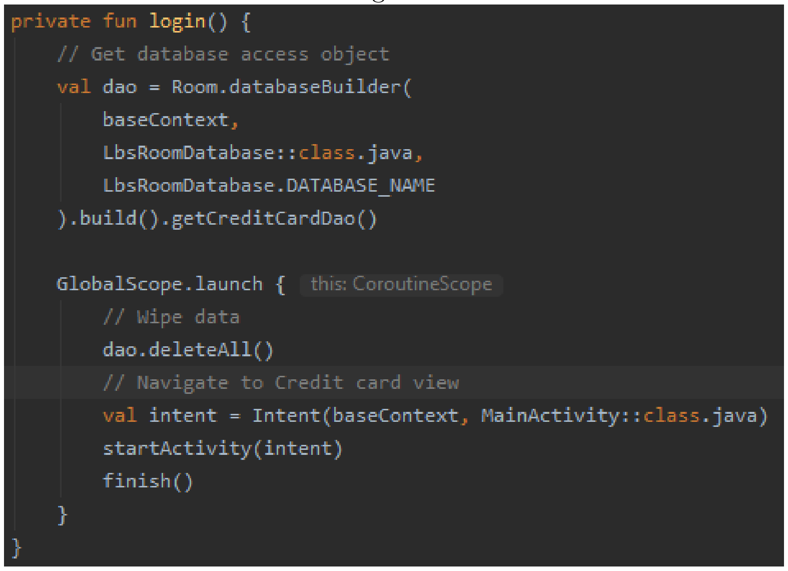


Figure 3: A visualization of the attack.

might rely on some of the data that got deleted for logged-in users.

Another way to solve this would be to also wipe the sensitive data on login to make sure that it's deleted. This solution was implemented in the prototype with the code shown in the image below.



```
private fun login() {
    // Get database access object
    val dao = Room.databaseBuilder(
        baseContext,
        LbsRoomDatabase::class.java,
        LbsRoomDatabase.DATABASE_NAME
    ).build().getCreditCardDao()

    GlobalScope.launch {  this: CoroutineScope
        // Wipe data
        dao.deleteAll()
        // Navigate to Credit card view
        val intent = Intent(baseContext, MainActivity::class.java)
        startActivity(intent)
        finish()
    }
}
```

Here the app again uses the DAO to delete all data, once the data is deleted it lets the user navigate to the credit card view.

A third option would be to have two logged-in flags, one declaring that a logout has begun and one to keep track of the logged-in state. The first should be set once the logout starts and the second when it finishes. Then if the app is ever in a state where the first flag indicates that a logout has begun but the logged in flag states that the user is still logged-in, the app can try to log out again before showing any data to the user. This method prevents some unnecessary data wipes but also adds more complexity to the code and was therefore not implemented.

5.2 Backup and Unencrypted Data

According to the 3rd section, the way that the application stores data does not include any encryption, so any data, both in the shared preferences and the SQLite database, will be stored as plain text, the way they were input by the user, if that data is sensitive, and the attacker was able to somehow access the storage, that will expose everything stored by the application.

This is based on OWASP's Top 10 Mobile Risks for 2014 and 2016, where Insecure Data Storage is 2nd on the list [15][14], and the attack itself did not follow a one specific source but rather a collection of information and tools, where we learned how and where applications store data, conditions to access it, and the tools needed.

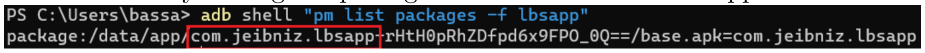
5.2.1 Attack

Note: As we do plan on attacking the application and its method of storing data, and not the Android OS itself, these attacks have to make some assumptions, in some cases, we assume that the phone is not locked, and another method of the attack assumes that the phone is rooted [19].

For this attack, we used the following tools: **Android Debug Bridge** [3] which lets us communite with the device through a command line, **Android Backup Extractor** [2] to read data from an android backup, **DB Browser for SQLite** [17] in order to read an SQLite file, and **7zip** [16] to unpack the extracted files.

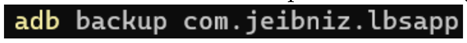
This first attack assumes that the phone is unlocked.

We first start by finding the package name of the vulnerable application:



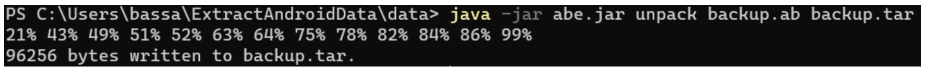
```
PS C:\Users\bassa> adb shell "pm list packages -f lbsapp"
package:/data/app/com.jeibniz.lbsapp-rHtH0pRhZDfpd6x9FPO_0Q==/base.apk=com.jeibniz.lbsapp
```

we then use adb's backup command to get a backup of the application's data :



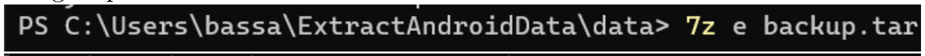
```
adb backup com.jeibniz.lbsapp
```

which will generate a backup.ab file in the current directory, we are able to extract data from that file using the android backup extractor tool, the tool is given the name of the backup file and the name of the output compressed file:

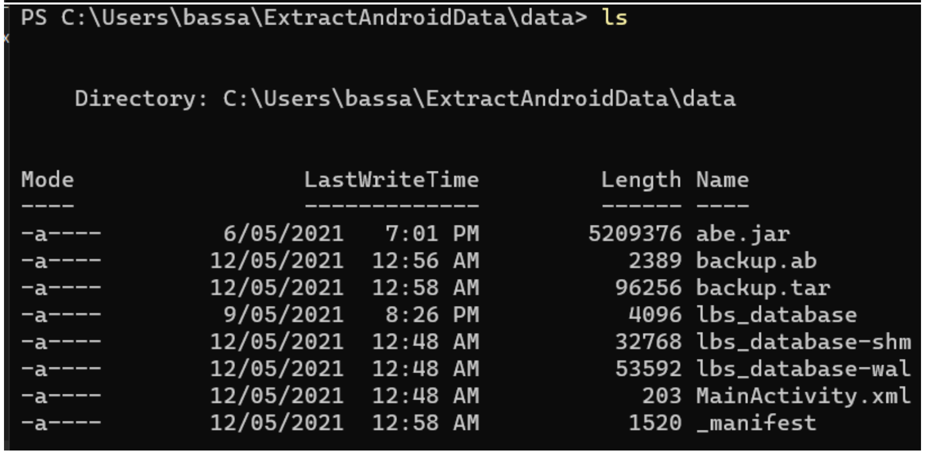


```
PS C:\Users\bassa\ExtractAndroidData\data> java -jar abe.jar unpack backup.ab backup.tar
21% 43% 49% 51% 52% 63% 64% 75% 78% 82% 84% 86% 99%
96256 bytes written to backup.tar.
```

using 7zip:

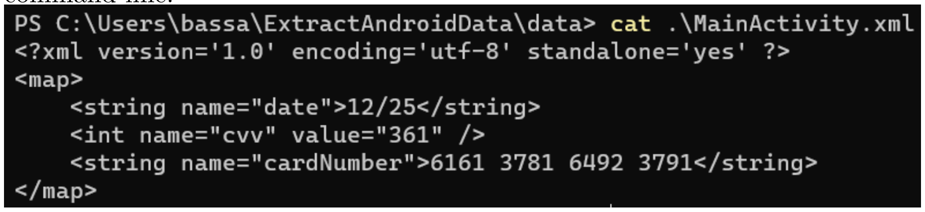


```
PS C:\Users\bassa\ExtractAndroidData\data> 7z e backup.tar
```



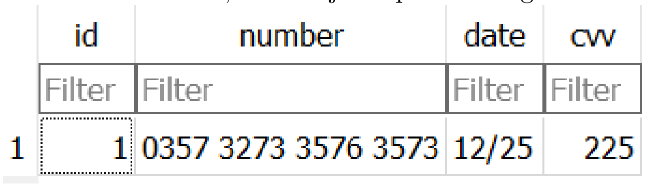
```
PS C:\Users\bassa\ExtractAndroidData\data> ls


    Directory: C:\Users\bassa\ExtractAndroidData\data


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         6/05/2021   7:01 PM        5209376 abe.jar
-a----        12/05/2021  12:56 AM           2389 backup.ab
-a----        12/05/2021  12:58 AM          96256 backup.tar
-a----         9/05/2021   8:26 PM           4096 lbs_database
-a----        12/05/2021  12:48 AM          32768 lbs_database-shm
-a----        12/05/2021  12:48 AM          53592 lbs_database-wal
-a----        12/05/2021  12:48 AM            203 MainActivity.xml
-a----        12/05/2021  12:58 AM           1520 _manifest
```

now we have the MainActivity.xml (shared preferences) and the lbs-database (SQLite database), you can read the .xml file using any xml file, or through the command line:



```
PS C:\Users\bassa\ExtractAndroidData\data> cat .\MainActivity.xml
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <string name="date">12/25</string>
    <int name="cvv" value="361" />
    <string name="cardNumber">6161 3781 6492 3791</string>
</map>
```

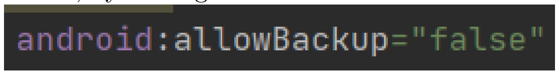
and there’s our data inside the xml tags, just as the user input them.
as for the database, we will just open it using the DB Browser for SQLite:



	id	number	date	cvv
	Filter	Filter	Filter	Filter
1	1	0357 3273 3576 3573	12/25	225

5.2.2 Mitigation

A countermeasure for this attack would be to not allow backups for your application, by editing the manifest file:



```
android:allowBackup="false"
```

In some cases, this is sufficient, since not all applications store data that need to be backed up.

Another countermeasure that is more ideal is to encrypt the data, that way, the data is no longer stored in plain text, so even if the attacker gets a hold of the storage, the data would not be intelligible to them.

We have included our implementation of an encryption/decryption method at the end of the report, but note that even tho our approach increases security, it is still vulnerable. Encrypting and decrypting locally means that the key for decryption is also stored locally, thus rendering it up for grabs for an advanced attack, Android helps with securing the key by providing a keystore [4]. Ideally, the decryption would happen on a remote server.

Some libraries are available that provide tools for encrypting local data, such as Encrypted Shared Preferences [8] and SQLCipher [13] are available out there to help with local encryption.

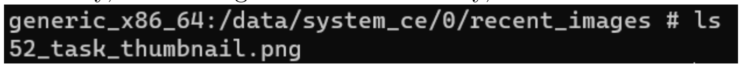
5.3 Recent Apps Image Preview

One way the Android OS improves user experience is by preserving the state of recently opened application, that way, the user can pickup where they left off if they switch applications, what it also does is that it takes a screenshot of the last state you left the activity in, that way you can preview the state while switching applications, a security threat ensues when the screenshot contains sensitive information, this screenshot is usually secure inside the system files, but in some cases, if the device is rooted [19], this screenshot can be accessed.

This exploit is related to the exploit in section Backup and Unencrypted Data as in it requires access to the storage. This vulnerability is found in 65% of mobile applications according to Positive Technologies [18], the attack and mitigation are also built on their description of this exploit.

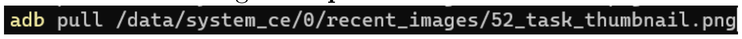
5.3.1 Attack

For this attack, we are going to be using Android Debug Bridge [3] again, we are also making an additional assumption that the device is rooted.
We will first enter adb's root mode using the **adb root** command, and then connect to the device, the screenshots are stored inside the system files, usually different for every device, in our case, it's stored inside the /data/system_ce/0/recent_images directory, if we navigate to that directory, we can find all of the thumbnails:



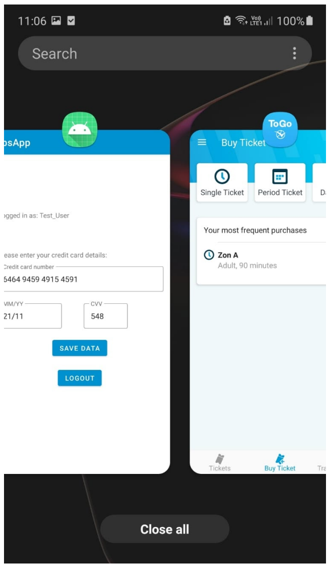
```
generic_x86_64:/data/system_ce/0/recent_images # ls
52_task_thumbnail.png
```

We can now our thumbnail, we just need to pull it to our device and view it, we will do that using adb's **pull** command:



```
adb pull /data/system_ce/0/recent_images/52_task_thumbnail.png
```

The image is now on our device and can be viewed using any image viewer:



ogged in as: Test_User

ease enter your credit card details:

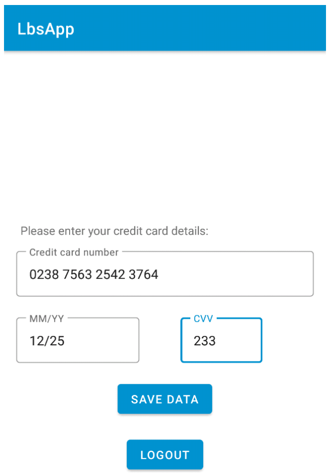
Credit card number: 5464 9459 4915 4591

MM/YY: 21/11

CVV: 548

Your most frequent purchases

Zon A
Adult, 90 minutes



Please enter your credit card details:

Credit card number: 0238 7563 2542 3764

MM/YY: 12/25

CVV: 233

5.3.2 Mitigation

The mitigation for this attack is quite simple, you are able to tell the OS that this window contains sensitive data, and that it should not store a screenshot of this activity's state, this is done through a security flag tagging the activity as secure:

```
window.setFlags(
                WindowManager.LayoutParams.FLAG_SECURE,
                WindowManager.LayoutParams.FLAG_SECURE)
```

Now if we attempt the attack again, the recent_images directory will no longer contain a screenshot of our activity, and the application's preview image will be replaced with a blank screen.

6 Evaluation and Conclusion

Looking at the possible exploits above, one can say that it is hard to get data from local storage but far from impossible. With regard to how sensitive some of the data can be, Android developers should some precautions to make it harder for individuals with malicious intents to extract data.

For most of the attacks to work, the device has to be unlocked. Therefore, a good takeaway from this report is how important a good lock on a device can be. This report highlights the fact that the consequence of someone gaining access to an unlocked device can be much more severe than many people might think. Another factor that made attacks easier to execute was that the **allow**

debugging option was activated. However, the effect of this was far less significant. Lastly, rooted devices allow adb access to all files in the device which makes the data stored on that device very hard to protect (if not impossible), therefore another takeaway is that data stored on a rooted Android device is not secure.

An important point to mention is that the found vulnerabilities are not vulnerabilities in Android but rather a combination of mistakes from developers and users getting either not sufficiently protecting their devices or giving an attacker permissions that they should not have.

Regarding encryption, one has to remember that it is a trade-off. Encrypting data will make it harder to access but it will also increase the read and write times to memory which might affect the user experience in the end. It will also increase the complexity of the codebase making it more prone to bugs and taking more resources to develop. Because of this, it is important to decide what to encrypt and what to leave as plain text. It is also important to realize that data stored locally can never be fully secure since the instructions of how to access it has to be stored in the code which can be decompiled by an attacker. It is therefore up to the developer to ether not store very sensitive data on the device or at least making it hard enough to access to that it is not worth the effort to access it.

References

[1] Haya Altuwaijri and Sanaa Ghouzali. “Android data storage security: A review”. In: *Journal of King Saud University-Computer and Information Sciences* 32.5 (2020), pp. 543–552.

[2] Nikolay Elenkov. *Android Backup Extractor*. URL: https://github.com/nelenkov/android-backup-extractor#readme.

[3] Google. *Android Debug Bridge, Android developers*. URL: https://developer.android.com/studio/command-line/adb.

[4] Google. *Android keystore system, Android developers*. URL: https://developer.android.com/training/articles/keystore.

[5] Google. *Android Studio*. URL: https://developer.android.com/studio/releases.

[6] Google. *Data and file storage overview, Android developers*. URL: https://developer.android.com/training/data-storage.

[7] Google. *Default exception handler, Android documentation*. URL: https://developer.android.com/reference/java/lang/Thread.UncaughtExceptionHandler.

[8] Google. *EncryptedSharedPreferences, Android developers*. URL: https://developer.android.com/reference/kotlin/androidx/security/crypto/EncryptedSharedPreferences.html.

[9] Google. *Jetpack, Android developers*. [Online; accessed 11-May-2021]. URL: https://developer.android.com/jetpack.

[10] Google. *Room, Android developers*. [Online; accessed 11-May-2021]. URL: https://developer.android.com/training/data-storage/room.

[11] Google. *Shared prefrences, Android developers*. [Online; accessed 11-May-2021]. URL: https://developer.android.com/training/data-storage/shared-preferences.

[12] GuardTools. *GuardTools Mobile*. URL: https://play.google.com/store/apps/details?id=com.guardtools.android&hl=sv&gl=US.

[13] Zetetic LLC. *SQLCipher*. URL: https://github.com/sqlcipher/android-database-sqlcipher.

[14] OWASP. *M2: Insecure Data Storage*. URL: https://owasp.org/www-project-mobile-top-10/2016-risks/m2-insecure-data-storage.

[15] OWASP. *OWASP Mobile Top 10*. URL: https://owasp.org/www-project-mobile-top-10/.

[16] Igor Pavlov. *7zip*. URL: https://www.7-zip.org/.

[17] Mauricio Piacentini. *DB Browser for SQLite*. URL: https://sqlitebrowser.org/.

[18] Positive Technologies. *Vulnerabilities and threats in mobile applications, 2019*. URL: https://www.ptsecurity.com/ww-en/analytics/mobile-application-security-threats-and-vulnerabilities-2019/.

[19] Wikipedia. *Rooting (Android)*. URL: https://en.wikipedia.org/wiki/Rooting_(Android).

Appendix A: Contributions

Both authors have worked on each part of the projects together.

Appendix B: Cryptography

```
class Cryptography {
    fun generateSecretKey() {
        // Define key specs
        val spec = KeyGenParameterSpec
            // Name of the the key in the storage and state that is it
            // for encryption/decryption
            .Builder(KEY_ALIAS, KeyProperties.PURPOSE_ENCRYPT
                    or KeyProperties.PURPOSE_DECRYPT)
            // Set block mode for symmetric encryption, CBC
            .setBlockModes(KeyProperties.BLOCK_MODE_CBC)
            // Padding schema, PKCS7
            .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_PKCS
            7)
            // Key size = 128
            .setKeySize(128)
            // Create an instance of KeyGenParameterSpec
            .build()
        // Create a KeyGenerator object made for AES Algorithm with the
        // name of the key provider
        val keyGenerator =
        KeyGenerator.getInstance(KeyProperties.KEY_ALGORITHM_AES,
                                                    KEY_STORE)
        // Initialize the key generator with spec parameter sets
        keyGenerator.init(spec)
        // Generate a key, it will be automatically stored in KEY_STORE
        keyGenerator.generateKey()
    }

    private fun getSecretKey(): SecretKey {
        // Get KEY_STORE keystore instance, with null password
        val keyStore = KeyStore.getInstance(KEY_STORE).apply {
        load(null) }
        // Fetch the key by its name, with null password
        val secretKeyEntry = keyStore.getEntry(KEY_ALIAS, null) as
        KeyStore.SecretKeyEntry
        // Return key
        return secretKeyEntry.secretKey
    }

    fun makeAes(rawMessage: ByteArray, cipherMode: Int): ByteArray? {
        return try {
            // Create a cipher that applies INSTANCE_MODE transformation
            // to the input
            val cipher: Cipher = Cipher.getInstance(INSTANCE_MODE)
            val output: ByteArray

            // If cipherMode is set to decryption
            if(cipherMode == 2) {
                // Get the IV from the input
                val ivParameterSpec =
                IvParameterSpec(rawMessage.take(16).toByteArray())
                // Initialize the cipherMode with decryption, secretkey,
                // and the IV
                cipher.init(cipherMode, this.getSecretKey(),
                ivParameterSpec)
                // Return the last 16 bytes, which include the message,
                // first 16 are the IV
                output = cipher.doFinal(rawMessage.copyOfRange(16,
                rawMessage.size))
                output
            // If cipherMode is set to Encryption
            } else {
                // Initialize with encryption and our secret key
                cipher.init(cipherMode, this.getSecretKey())
                // Encryption
                output = cipher.doFinal(rawMessage)

                // Putting the IV at the beginning of the message for
                // later decryption
                var outputStream = ByteArrayOutputStream()
                outputStream.write(cipher.iv)
                outputStream.write(output)
                // Converting to ByteArray and outputing
                outputStream.toByteArray()
            }
        } catch (e: Exception) {
            e.printStackTrace()
            null
        }
    }
    companion object {
        const val KEY_ALIAS: String = "demoKey"
        const val KEY_STORE: String = "AndroidKeyStore"
        const val INSTANCE_MODE: String = "AES/CBC/PKCS7Padding"
    }
}
```