Android Application Vulnerability Scanners

Dimitris Vagiakakos (E18019)
Department Of Digital Systems
University Of Piraeus
Piraeus, Attiki, Greece
dimitrislinuxos@protonmail.ch

Stavros Gkinos (E18043)
Department Of Digital Systems
University Of Piraeus
Piraeus, Attiki, Greece
stgkinos@protonmail.com

Ioannis Karvelas (E18066)
Department Of Digital Systems
University Of Piraeus
Piraeus, Attiki, Greece
jonncarv@gmail.com

ABSTRACT

Android, nowadays, is the most common mobile Operating System that is found in more than 2.8 billion smartphones and portable devices. Being such a common OS it has attracted various attackers trying to exploit any vulnerability that can be found. Especially, nowadays in the meta-Covid-19 period and contact-less transactions are becoming more of an everyday routine, it is a must that Android's safety taken into consideration from big companies and of course from independent everyday users.

1 INTRODUCTION

In our days, most of the people own at least one smartphone. In the end of 1990s, a growth in use of PDAs (Personal Digital Assistant) was obvious. As the time was passing by, these PDAs transformed into mobile devices, mainly known as smartphones. These devices have the abilities of a Personal Computer and a variety of features such as connectivity, storage, multi-tasking, bio-metrical sensors (such as fingerprint sensor), wireless charging and an advanced graphical interface that makes the usage accessible to everyone. This time, there are 3.5 billion smartphone users out there by the number being increased day by day. Except smartphones, today, there are a lot of other smart devices that make people's every day life easier, using the same Operating System as smartphone users. There are devices such as smart watches,bands, refrigerators, coffee machines, cameras being a few examples, that use operating systems like iOS, Android and Windows. The most common open source operating system running in mobile devices is Android. Our years are known as the Internet of Things (IoT) years because many devices are connected to one or more networks. As previously stated, these devices assist us every day while they wait for our orders. However, as Android is a mainstream operating system and especially due to Covid-19 nowadays we use smart devices for banking applications. These devices can be vulnerable to malicious attacks and as the Android popularity is increasing, the probability will become higher and higher for more security attacks. Android is an Open-Source OS and developers prioritize consumer demand over security. Having this information in our minds, combined with the device's sensitivity in malicious attacks, is lethal for users' data. In this paper most know security application vulnerabilities, how Android works in terms of security and privacy and vulnerability scanners will be discussed and presented.

2 THE NEED OF MOBILE SECURITY

Mobile security is a developing field of study. Many studies that are working on mobile security have been published in the last two to three years, not only for Android but also for iOS. The character (open source) and the extensibility of Android made it popular for mobile devices and a strong base for the IoT. Static or dynamic techniques are used in these frameworks. The key benefit of these methods is that they do not require any changes to the Android code, and the hardening tools may be easily installed as regular apps on the device. Taint assessments on developing or developed applications can address these issues. One of the most recent malware attacks in Android was the CovidLock ransomware back in 2020. As the coronavirus was declared as a pandemic, some people tried to take advantage of the uncertainty around it to attack plenty of users. The application was an Android ransomware that locked out the victim and was asking, the victim, for money as a ransom to grant him the control back. Though the program claims to provide COVID-19 heat map visualizations and other statistical data, it is a ransomware distributor. The software tries to persuade the user to grant it administrator access by offering various types of information during installation. With that access, it can then encrypt all contacts, photos, videos, and social media accounts unless the user agrees to pay a Bitcoin ransom. The attacker threatens to reveal all sensitive information and wipe the phone's memory if the ransom is not paid.

2.1 Malware Analysis on Android

Detecting and forcing to stop threats is known to be a very strenuous task. Static Analysis can be used to identify threats and then fix the security holes. To be more precise, it examines an application's

code or source code for harmful activities without requiring it to be executed. It entails extracting information from bundled APKs, such as source code. DroidMat, dex2jar, Procyon, being a few examples, are tools were utilized to dissemble analysis tool. Android application run within the Android framework. This creates imposing of complex life-cycle on the apps, invoking call-back methods pre-defined or user-defined different times during execution of app. The tool must predict apps control flow and hence requires precise life-cycle modeling. Imprecise life-cycle modeling will cause the tool to lose or overshadow important data flows. Static Analysis tools have several approaches depending upon precision, runtime, scope, and focus.To give an example, QARK (Quick Android Review Kit) is a tool that detects a frequent vulnerability in an Android application. It differs from most typical tools in that it will highlight vulnerabilities as well as include adb commands. QARK is checking for weak or improper cryptography, incorrect x.509 certificate validation, data leakage activities, sending of insecure Broadcast Intents, insecurely constructed Pending Intents, and a variety of other security flaws. On the other hand, Dynamic Analysis (Dynamic data flow analysis) of an app links testing and evaluation by sending data to an application in real time in a controlled environment. It is well known for analyzing and monitoring the situation behind the scenes the time that the application is running. Although tools of static analysis are very quick, they fail against encrypted and transformed malwares. More about these will be discussed in the next sections.

3 THE ANDROID ARCHITECTURE

Now it's time to mention the main Android architecture and some of its features, especially the system security features and stricture. Android Architecture is split up into 4 main layers:

(1) Applications: On this layer, we meet the Android Applications. Android Operation System uses are using Android Package format with the file extension apk. Android Applications are written in Java or Kotlin and installed to /system/apps for system applications and /data/apps for user applications.

(2) Application Framework: Developers get complete access to the framework APIs that are used by apps. The architecture is designed to facilitate component reuse; any program can publish its capabilities, which can subsequently be used by other applications (subject to safety rules framework). The user can replace components using the same technique. As a result, the applications that can be developed are limitless.

(3) Libraries: Android provides several C/C++ and Java libraries that are utilized by various Android components. The Android application framework exposes several functionalities to developers, including a System C library (C standard library implementation), media libraries, graphics libraries, 3d support via Open GL, SQLite embedded database, and so on.

(4) Android Runtime: Android includes a set of libraries that provide many of the capabilities found in the Java language's libraries. Every Android app operates its own process, with its own Dalvik Virtual Machine instance (DVM).

(5) Linux Kernel: Developing devices drivers is similar to developing a typical Linux device driver. Android uses its own harden version of Linux Kernel with special extended features which are extremely useful for a mobile embedded platform. For instance, Android uses a memory management system that is more aggressive in preserving memory, power manager system services called wake locks which prevent the system to close unused applications. Android includes a set of foundation libraries that provide most of the capabilities found in the Java language's libraries.

3.1 Android APIs

API Level is an integer value that uniquely identifies a version of the Android platform's framework API revision. Applications can use the framework API provided by the Android platform to interface with the underlying Android system. The framework API is made up of the following parts:

(1) A core set of classes and packages

(2) For declaring a manifest file, a set of XML elements and properties is used.

(3) A set of XML elements and attributes that can be used to declare and access resources.

(4) A set of Intents

(5) A set of permissions that apps can request, as well as the system's permission enforcement.

It should be mentioned that The Android application framework API that it provided can be updated with each new version of the Android platform. The framework API is updated in such a way that the new API is backwards compatible with previous versions. That is, the majority of API modifications are additive, meaning they provide new or replace existing functionality. As parts of the API are upgraded, older parts that have been replaced are deprecated but not removed, allowing existing applications to continue to use them. Parts of the API may be modified or eliminated in a small number of circumstances, but such changes are usually only required to ensure API robustness and application or system security. All other API parts from previous releases are carried through unchanged.

3.2 Android Keystore System

The Keystore system is used by the KeyChain API, introduced in Android 4.0 (API level 14); the Android Keystore provider feature, introduced in Android 4.3 (API level 18); and the Security library, available as part of Jetpack. This document goes over when and how to use the Android Keystore provider.

In terms of security features, Android Keystore system safeguards key material against unauthorized use. To begin, Android Keystore prevents unauthorized use of key material outside of the Android

device by preventing key material extraction from application processes and the Android device as a whole. Secondly, Android KeyStore reduces unauthorized key material use on the Android device by requiring apps to specify authorized key uses and then enforcing these restrictions outside of the apps' processes. Two security measures are used to protect the key material of Android Keystore keys from extraction:

(1) The application process never includes any key material. When an application performs cryptographic operations with an Android Keystore key, plaintext, ciphertext, and messages to be signed or verified are fed behind the scenes to a system process that performs the cryptographic operations. If the app's process is compromised, the attacker may be able to use the app's keys but will be unable to extract the key material (for example, to be used outside of the Android device).

(2) Key material may be bound to the Android device's secure hardware (e.g., Trusted Execution Environment (TEE), Secure Element (SE)). When a key has this feature enabled, its key material is never exposed outside of secure hardware. If the Android OS is compromised or an attacker gains access to the device's internal storage, the attacker may be able to use any app's Android Keystore keys on the device but not extract them. This feature is enabled only if the secure hardware of the device supports the specific combination of key algorithm, block modes, padding schemes, and digests that the key is authorized to use. To determine whether a feature is enabled for a key, obtain a KeyInfo for the key and examine the KeyInfo return value. isInsideSecurityHardware(). Hardware Security Module: StrongBox Keymaster, an implementation of the Keymaster or Keymint HAL that resides in a hardware security module such as a secure element, can be installed on supported devices running Android 9 (API level 28) or higher. While Hardware security modules can refer to a variety of key-storage implementations where a linux kernel compromise will not reveal them (for example, TEE), Strongbox specifically refers to devices such as embedded Secure Elements (eSE) or on-SoC secure processing units (iSE).

(1) CPU

(2) Secure Storage

(3) Secure Timer

(4) AP reboot notification pin

(5) True random-number Generator

(6) Additional safeguards against package tampering and unauthorized app side-loadings.

3.3 Android Security Patches

Android Open Source Project releases monthly security updates for the Android. Monthly device updates are an important tool for keeping Android users safe and their devices to stay secure. The public bulletin contains fixes from a variety of sources, including the Android Open Source Project (AOSP), the upstream Linux kernel, and system-on-chip (SOC) manufacturers. Manufacturers of devices:

(1) Android platform fixes are merged into AOSP within 24–48 hours of the security bulletin's release and can be downloaded directly from there.

(2) SOC manufacturer fixes are available directly from the manufacturers.

It should be mentioned here that Google provides Google Play Security Patches for Google Certified devices, which include some security fixes from from the main security patches. These kind of updates can replace some of security patch fixes easily from Google directly without any device manufacturer and maintainer needed, but not the entire ROM's problems or vendor bugs.

3.4 Android Debug Bridge

The Android Debug Bridge (adb) is a powerful command-line tool for interacting with Android devices. The adb command allows users to do a variety of device tasks, such as installing and debugging apps, as well as access to a Unix shell from which you they can run a variety of commands on the device. When user starts an adb client, it checks to see if there is already an adb server process running. If there isn't one, the server process is started. When the server boots up, it binds to local TCP port 5037 and listens for commands from adb clients—all adb clients connect with the adb server over port 5037. The server then establishes connections with all currently active connected devices. It looks for emulators by scanning ports in the range of 5555 and 5585, which is the range used by the first 16 emulators. The server establishes a connection to the port where it discovers an adb daemon (adbd). It's worth mentioning that each emulator has a pair of consecutive ports: one for console connections and the other for adb connections. It should be also mentioned that ADB supports connection and debugging using WIFI. This features we an extra implementation in custom ROMs until Android 11 when ADB over WIFI pushed to the main AOSP project. To be functional, ADB has three main components;

(1) A client which sends commands. The client runs on the

(2) A daemon (adbd) which runs commands on a device

(3) A server, which manages the communication between a client and the adb daemon. The server runs as a background process on the development machine.

Moreover, we can directly get shell from our computer to an Android device by using the command:

```
adb shell
```

Or we can just type the command we want after the adb command. Moreover, we can directly get shell from our computer to an Android device by using the command:

```
adb YOURLINUXCOMMAND
```

3.5 SELinux

Android implements Security-Enhanced Linux (SELinux) to enforce mandatory access control (MAC) across all processes, including those with root/superuser rights, as part of its own security policy (Linux capabilities). Bean Android can better secure and confine system services, control access to application data and system logs, minimizing the impacts of malicious software and secure users from, potential, code vulnerabilities on mobile devices running on SELinux. SELinux operates on the principle of default denial, which means that anything that is not explicitly permitted is refused.

SELinux can operate in two global modes:

(1) Permissive mode: In which permission denials are logged but not enforced. Permissive mode is used for debugging purposes and testing and it is not recommended for daily use as it is not enforcing policies.

(2) Enforcing mode: In which permissions denials are both logged and enforced, offering a more secure environment.

SELinux in enforcing mode and a matching security policy are included by default in Android. Disallowed actions are forbidden in enforcing mode, and all attempted violations are logged by the kernel to dmesg (dmesg creates kernel logs) and logcat (for application logs). Furthermore, SELinux features a per-domain permissive mode, which allows users to prefer which domains(processes) to make permissive while the whole of the system will remain in global enforcing mode. A domain is typically a label in the security policy that identifies a process or set of processes, with the security policy treating all processes labeled with the same domain in the same way. Per-domain permissive mode enables SELinux to be applied incrementally to a growing area of the system, as well as policy development for new services (while keeping the rest of the system enforcing).

SELinux came for the very first time in Android 4.3 Jelly at Permissive mode and in Android 4.4 for the very first time switcher to partially enforcing. In Android 5.0 and later, SELinux is fully enforced. After this significant change on Android Kernel, Android extended its sandbox features from the create of a unique Linux UID for each application at the time of application's installation to the enforcement on a limited set of crucial domains to more than 60 domains. Android 6.0 hardened the system by reducing the permissiveness of the policies to include better isolation between users and services. Furthermore, Android 6.0 implemented and divided permissions into Normal and dangerous permissions and user had the ability to accept and deny permissions to an application. Android 7.0 updated SELinux policies to further lock down the application sandbox and decrease attack surface. Android 8.0 updated SELinux to work with Treble which separates lower-level vendor code from the Android system framework. This release improved SELinux policy by allowing device manufacturers and SOC suppliers to update their parts of the policy, construct their images (vendor.img, boot.img etc.), and then update those images independently of the platform, or vice versa.

3.6 Permissions on Android

Permissions in applications assist users protect their privacy by restricting access to the following:

(1) Restricted data: Such as system's state and a user's contact information.

(2) Restricted actions: Such as connecting to a paired device and recording audio.

Determine whether you can receive the information or execute the activities without needing to declare permissions when your application delivers functionality that may require access to restricted data or restricted actions. Without declaring any permissions, you can fulfill various use cases in your app, such as snapping images, pausing media playback, and displaying relevant adverts.

Declare the required permissions if your application should access restricted data or conduct limited activities to meet a use case. When your program is installed, some permissions, known as install-time permissions, are automatically granted. Other permissions, referred to as runtime permissions, require your program to request the permission at runtime.

At first, Android application permissions are declared in the applications Android manifest xml file. Applications must be coded to ask as less permissions as possible to stay functional. If a permission is a runtime permission, user should be requested to grant permission at runtime.

Permissions on Android are divided into three categories:

(1) Install-time permissions

(2) Run-time permissions

(3) Special permissions

When the system grants an application a permission, the type of that permission indicates the extent of limited data that the app can access and the scope of restricted actions that the application can execute. Install-time permissions focus on providing the application with restricted access to restricted data and allow the application to run limited activities that have minimal impact on the system or other apps. When we define install-time permissions in an application, the system grants certain permissions to the app when the user installs it. Android includes several sub-types of install-time permissions, including normal permissions and signature permissions.

(1) Normal permissions: These permissions provide access to data and activities outside of application's sandbox. The data and activities, on the other hand, pose relatively minimal risk to the user's privacy and the functionality of other apps. Some of the normal permissions are:

   (a) BLUETOOTH

   (b) EXPAND_STATUS_BAR

   (c) GET_PACKAGE_SIZE

   (d) INSTALL_SHORTCUT

(e) INTERNET
(f) KILL_BACKGROUND_PROCESSES
(g) NFC
(h) READ_SYNC_SETTINGS
(i) READ_SYNC_STATS
(j) REORDER_TASKS
(k) REQUEST_IGNORE BATTERY_OPTIMIZATIONS
(l) REQUEST_INSTALL_PACKAGES
(m) SET_ALARM
(n) SET_TIME_ZONE
(o) SET_WALLPAPER
(p) VIBRATE
(q) WAKE_LOCK

(2) Signature Permissions:

Furthermore, it should be mentioned that if an application declares a signature permission that an another application has defined, and both applications are signed by the same certificate, the system grants the permission to the application that has been installed first. Otherwise, the permission will not be provided to the initial application.

(3) Special Permissions:

Special permissions are assigned to specific app operations. They can only be defined by the platform and OEMs. Additionally, when the platform and OEMs want to protect access to particularly powerful operations, such as drawing over other applications, they normally establish special permissions. A variety of user-toggleable procedures can be found on the Special app access page in system settings. Special permissions are used to implement many of these tasks. Some examples of special or dangerous permissions are:

(a) READ_CALENDAR
(b) WRITE_CALENDAR
(c) CAMERA
(d) READ_CONTACTS
(e) ACCESS_FINE_LOCATION
(f) READ_PHONE_STATE
(g) READ_PHONE_NUMBERS
(h) CALL_PHONE
(i) ANSWER_PHONE_CALLS
(j) READ_CALL_LOG
(k) WRITE_CALL_LOG
(l) PROCESS_OUTGOING_CALLS
(m) BODY_SENSORS
(n) READ_EXTERNAL_STORAGE
(o) WRITE_EXTERNAL_STORAGE
(p) ACCESS_MEDIA LOCATION
(q) ACCEPT_HANDOVER
(r) ACCESS_BACKGROUND_LOCATION

It should be mentioned that some custom Android implementations have some normal permissions as dangerous permissions and user should grant access to these permissions too. For example, CrDroid, an Android Lineage OS based custom ROM, supports to grant or deny internet access to each application.

3.7 Webview

Android System WebView is a system component that lets Android apps display web content inside them without opening a dedicated browser. In other words, Android System WebView is a web browser engine or an embedded web browser dedicated solely for applications to show web content. WebView objects allow programmers to include web material to their activity layout, but they lack some of the functionality of fully developed browsers. When applications require more UI control and complex configuration options, a WebView comes in handy. Webview allows to integrate web pages in a specially-designed environment to application and extend its features by using web applications. Older versions of Android had their own version of Android Webview integrated to the system. Especially, OEMs usually built their own versions of Android Webview (most of the times to integrate their own trackers). However, as practically Webview is a simpler limited web browser for applications, if OEMs didn't release system updates regularly, most of the times Webview became an attack surface as it ends up outdated with known security holes. On newer Android versions, Webview can be upgraded directly from Google Play Store from the OEM devices and in open-source community, there are multiple Open-Source Webview implementations without trackers that users can use and upgrade on a monthly basis. Examples of Open-Source Webviews are Bromite Webview, UnGoogle Chromium Webview, Brave Webview etc.

3.8 Logcat

Logcat is a command-line tool that dumps a log of system messages, such as stack traces when the device throws an error and messages produced with the Log class from your app. The system process logd maintains the Android logging system, which is a set of structured circular buffers. The system determines and defines the collection of available buffers. The most important are main, which contains the majority of application logs, system, which contains messages from the Android OS, and crash, which contains crash logs. Each log entry contains a priority (VERBOSE, DEBUG, INFO, WARNING, ERROR, or FATAL), a tag that identifies the log's origin, and the message itself. logcat can be called by using adb with the command:

```
adb logcat
```

3.9 Two-Factor Authentication

Two-factor authentication (2FA) requires from users to present two different kinds of information from two different sources to prove user's identity. When it comes to online accounts, three different types of IDs can be used for 2FA:

(1) Something only user should be aware of. In this case, a password, PIN, account number, street address, or even the last four digits of user's Social Security number will suffice.

(2) Something you can hold in your hands. This could be your phone, a key fob, or a USB security key.

(3) Bio-metrics can also be used but are not recommended as any leakage will end with access to our all sensitive information.

With so many online services available, it is of highly importance to have a unique password for each of these services. Duplicate passwords expose users to the risk of being hacked if someone obtains one password and then attempts to use it at multiple sites. It is impossible for a human being to remember all of these passwords to different websites and platforms, which is where password managers come in. The good news is that password managers have evolved so much that many of them now include extra features that are not only useful but also keep passwords and other information secure. Because of its simplicity, KeePassXC is currently the most famous open-source password manager which is available to all Operating Systems. Furthermore, applications as Aegis are 2FA applications which produces keys that work for only for a bit of seconds. However, is 2FA secure? Both yes and no. Using 2FA on an account is far more secure than not using it, but no security measure is perfect. Aside from that terrifying thought, using 2FA is usually sufficient protection for your "stuff," unless you are a high-profile target or extremely unlucky. On the plus side, if you are using 2FA and a phishing email tricks you into providing your password, they won't be able to access user's account. Most people use 2FA for online accounts by having a token sent to an application on their Android phones, and without that token, the email scammer will be unable to gain access. They will enter your account username or ID, then your password, and finally the token to proceed. Unless they have your phone, the effort required to circumvent the second ID requirement is enough to make the bad guy say "forget it!" and move on to someone else. Applications like Aegis or Google Authenticator are some examples of 2FA applications. Especially, Aegis encrypts the database with a different key from user's login unlock password for enhanced security.

3.10 Android Sandboxing

In general, to escape the Application Sandbox in a properly configured device, the security of the Linux kernel must be compromised. Individual protections enforcing the application sandbox, like other security features, are not invulnerable, so defense-in-depth is important to prevent single vulnerabilities from leading to compromise of the OS or other apps.

To enforce the application sandbox, Android employs several safeguards. These regulations have gradually been implemented and have significantly strengthened the original UID-based discretionary access control (DAC) sandbox. Previous Android versions included the following safeguards:

(1) SELinux introduced mandatory access control (MAC) separation between the system and apps in Android 5.0. However, because all third-party apps ran in the same SELinux context, UID DAC was primarily used to enforce inter-app isolation.

(2) The SELinux sandbox was extended in Android 6.0 to isolate apps across the per-physical-user boundary. Furthermore, Android has safer defaults for application data: The default DAC permissions on an app's home dir changed from 751 to 700 for apps with targetSdkVersion >= 24. This provided a more secure default for private app data (although apps may override these defaults).

(3) In Android 8.0, all apps were configured to use a seccomp-bpf filter, which limited the syscalls that apps could use, thereby strengthening the app/kernel boundary.

(4) All non-privileged apps with targetSdkVersion >= 28 in Android 9 must run in individual SELinux sandboxes, with MAC provided per-app. This protection improves app separation, prevents safe defaults from being overridden, and (most importantly) prevents apps from making their data world accessible.

(5) Apps in Android 10 have a limited raw view of the filesystem, with no direct access to directories such as /sdcard/DCIM called Scoped Storage. Apps, on the other hand, have complete raw access to their package-specific paths, as returned by any applicable methods, such as Context. getExternalFilesDir().

3.11 Best Practices

(1) Control: The user has control over the data that they share with apps.

(2) Transparency: The user understands what data an app uses, and why the app accesses this data.

(3) Data minimization: An app accesses and uses only the data that's required for a specific task or action that the user invokes.

As we can understand, applications must be coded from the developers to request the minimal number of permissions and request only the permissions that are really needed for the application to stay functional. Moreover, it is recommended for the permissions requests to be late into the flow of the application. For example, if we have a chatting application, we should consider that having microphone access all the time is not really needed.

4 DAMN INSECURE AND VULNERABLE WEB APPLICATION

Damn Insecure and Vulnerable App (DIVA) is a vulnerable Android application designed to be insecure to help the developers and security engineers to explore the Android Ecosystem and it flaws, vulnerabilities or bad coding practises they can be used and their

impacts to Android system security and information leakage. In the next sub-chapters we will play DIVA's challenges and we will demonstrate how we can take advantages of the vulnerabilities, the security flows and holes and we recommend best code practises.

4.1 Insecure Logging

On this challenge we have an insecure logging screen. Login screen asks the user to enter his credit card number. A bad implementation of logging screens can end up with the sensitive information leaked inside to the log files. Android uses logcat and logs almost everything on application activities. Let's enter some random data and hit 'Check out'. We can see an error message: "An error occured".

Seems interesting. An error message most of the time means that the application saved an error message to logs Let's check the logs. From adb we check the logs by entering:

```
adb logcat | grep -i diva
```

With this command, we have directly access to device's logs and by using grep, we can show up only logs that includes information with diva inside. We can find the credit card number as a plaintext inside in logs. The logging screen is damn insecure.

Let's check it's code to learn more about the flaw.

Opening up LogActivity.class we can see :

```
log.e(diva-log, Error while processing transaction
    ↪ with credit card:  + localEditText.getText().
    ↪ toString());
```

So, literally, because of the error, the program with using the getText().toString() passes the sensitive information directly as plaintext in the logs. It must be removed and we will no longer have leaks in the log file.

This security issue looks dumb, however is one of the most common security flaws happen in applications and no one notices. For example, MicroG, an open source implementation of Google Play Services has this security leakage bug for more than 3 years when a device's ROM worked in debugging mode. As a consequence, when the user tried to login to his own Google Account, its Gmail and password ended up to the logs of the device. As most of the MicroG users were using Official Stable Custom ROMs and not experimental builds, no one noticed the bug until September of 2021 on a Telegram channel.

4.2 Android Data Storage

On this challenge we have a login screen which asks the user about a username and a password. Generally speaking, Android saves files in Android/data folder, on a different folder of each application. For example, DIVA saves our files in the folder jakhar.aseem.diva in Android/data folder. Applications are forbidden of access except on their own folder in Android/data. However, except if user gives a special data accesses permission, where then are able to have accesses in all data except the Android/data folder of other applications. That means that is almost impossible without an exploit to have access to Android/data folder. But what happens if we have root access? Let's take a root shell in our device:

```
adb shell
su
```

now we have root permissions and we have access to all user and system files. Let's check the Android/data/jakhar.aseem.diva now:

```
cd Android/data
ls | grep diva //to show us exactly the name of the
    ↪ diva's folder
cd jakhar.aseem.diva
```

we can see 3 folders, "cache","code_cache","databases", "shared_prefs". Let's check the folder "shared_prefs" by typing:

```
cd shared_prefs
ls
```

Let's see what's inside the file: "' cat FILEs_NAME "' And we can see our username:password unecnrypted saved in Android/data. We should never save sensitive files or information unencrypted, even if Android protects the Android/data file access from other applications and the user. A simple flaw to grant root access or bypass the system protection and anyone can read the sensitive information. It should be mentioned here that most of the Android implementations, especially these from OEMs does not provide root access out of the box and user has to unlock his device bootloader at first and then grant root access to the device. That means it is as an Android gets every month its security patches is very difficult to earn root access with locked bootloader.

Moving on the next challenge, the second part of Insecure Data Storage is almost the same with the first part, but on this part we have to execute an SQL injection. We gain again root access by using the su command and then we are going to Android/data/-jakhar.aseem.diva/databases, On this folder if we type ls we can see some SQLite databases. Let's check ids2 db:

```
sqlite3 ids2
.tables
```

It shows us 2 tables: * android_metadata * myuser Let's send the SQL query:

```
select * from user;
```

and it sends back the username|password . This vulnerability is very common to smart-band applications too. For example, following this methodology intruders can gain the private key a smart-band and then add it to an another device. Then, he can get all of user's sensitive information for his band very easily!!!

In the next Data storage challenge we will discuss about temporal files. Android, as other Operating Systems support temporal files in order to save information for temporal use. After system's reboot, temporal files are being deleted by default. However, as the devices is powered on if we save sensitive information as a temporal file

unencrypted, a malicious attacker can easily read the file and get the information that it should have been secure and private. So, we have a basic login screen again. We input our credentials and we click the Save button.Let's find out where the credentials are stored. At first, let's check the code of the third Data Storage challenge. We can see that it saves the credentials to a temporary file:

So we now that the credentials are stored on a file that starts with "uinfo". We search recursively in all DIVA's data folder:

```
cd Android/data
cd jakhar.aseem.diva
find . -name 'uinfo*'
cat uinfo2030199978tmp
```

And we can see the credentials in plain text. We should never save anything in storage unencrypted. These were only a few examples of bad security practises and we saw that there are many commercial and open-source projects with security leaks as these we describe above. For these reasons we should be very cautious when we download and use third party applications.

4.3 Access Control

On this challenge main task is to open and be able to get the API Credentials by externally calling the activity, in this case, meaning without pressing the VIEW API CREDENTIALS Button. Now it is important to mention and explain what AndroidManifest.xml is. To begin with, every single Application specifies inside AndroidManifest.xml App's permissions as well as the App's activities and how they can be called, for example, either from CLI or GUI through a button being pressed. In order, now, to open the API Credentials we need to view the content of DIVA's AndroidManifest.xml. Now that we are able to see what is inside DIVA's AndroidManifest.xml, we can notice that there is an action VIEW_CREDS named as:

```
jakhar.aseem.diva.action.VIEW_CREDS
```

We have to find a way to run this action separately. In order for this to be done we use adb commands.

```
adb shell am start -a jakhar.aseem.diva.action.
    ↪ VIEW_CREDS

am: activity manager
```

As a result we are able to see on the smartphone's screen the API Credentials. Concluding, we can understand why and how much importance is for an app to protect properly it's components from other apps and users.

4.4 HardCoding

On this challenge our main task to insert the right vendor key in order to grant access. Many developers, in order to make their lives easier, are hardcoding information. This action, as we will see later, leads to the conclusion that everything inside an app must well protected. So, we have to see the code of this challenge. By searching we see that there is a file inside Diva's directory named:

```
HardcodeActivity.java
Which contains:
cat HardcodeActivity.java
```

Looking closely at the public void access view we can see that the developer has hardcoded stored the correct vendor key which is

```
"vendorsecretkey"
```

By inserting this vendor key we have granted access.

5 ANDROID VULNERABILITY SCANNERS

Vulnerability scanners are useful tools for discovering and reporting on known vulnerabilities in an organization's IT infrastructure. Using a vulnerability scanner is a simple yet important security strategy which could benefit any organization. These scans can help an organization identify what security threats they may be up against by revealing potential security flaws in their surroundings. Many businesses use numerous vulnerability scanners to guarantee that every asset is fully covered, resulting in a complete image. Many various scanners have been developed throughout the years, each with its own set of options and functions. There are both commercial and open source vulnerability scanners which can support developers to identify security flaws to their applications. Furthermore, online vulnerability scanners are available where developers can upload their applications to be checked for security flaws and vulnerabilities. For example, QUIXXI Security offers an online mobile application vulnerability scanner. Quixxi supports free vulnerability check to developers and then it offers for free the vulnerability assessment. For fixing recommendations, developers have to subscribe to their premium service. Other online Android vulnerability scanners are App-Ray, ImmuniWeb, Astra Penetest etc. Except the commercial Android App Vulnerability Scanners, as Android is an mainstream Open-source mobile Operating System, open-source community created various automated mobile security frameworks and application vulnerability scanners. Mobile Security Framework, StaCoAn and Quick Android Review Kit (Qark) are only a few examples of open-source tools that are getting updates very often. Of course there are much more except these tools, however most of the vulnerability scanners are out-dated and are focused to older Android versions.

5.1 Quick Android Review Kit

Quick Android Review Kit (Qark) is a Python3 based tool which is designed to check for a variety of security vulnerabilities in Android applications, either in source code or bundled APKs. The application can also generate deployable "Proof-of-Concept" APKs and/or ADB commands which can exploit several of the vulnerabilities it discovers. Since this tool concentrates on vulnerabilities that can be exploited under otherwise secure conditions, there is no need to root the test device. Qark is one of the most alive open-source vulnerability scanners. Developers who want to check their apk files the only thing they have to do is to install qark and then enter:

```
qark --apk APPLICATION_NAME.apk
```

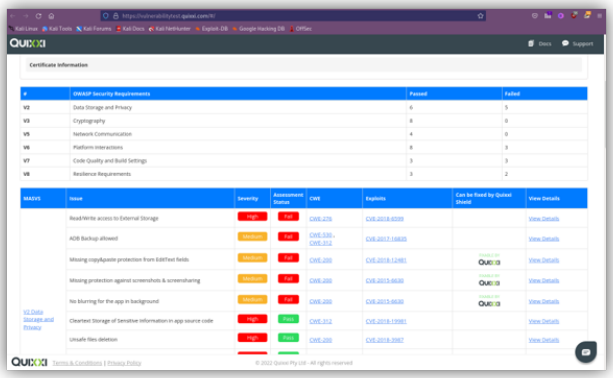

Figure 1: Quixx Overview - An online Android Application Vulnerability Scanner

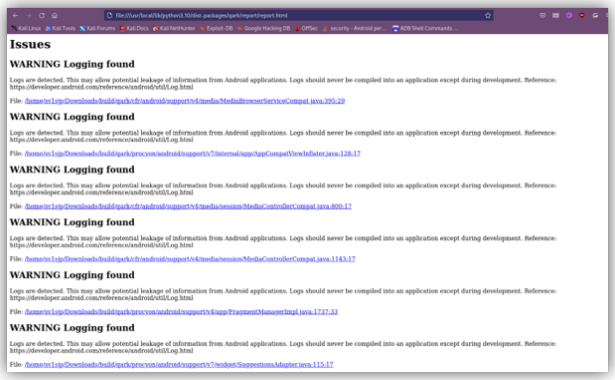

Figure 2: Qark' HTML File with all the Potential Security Issues

Then, Qark will check the application and create a details .html file with all potential issues, the line of the security flows and some ideas and information to fix the security flaws. Qark is extremely easy to use and as it is written in Python3, is available in all Operating Systems.

6 CONCLUSION

This paper has only a few examples of vulnerabilities that can be found in Android Applications and can end up fatal. It is of high importance to write secure code, use secure code practices and if it is possible to use libraries instead of reinventing the wheel. A library that is broadly used likely has been audited by multiple members of the community, and so it has a better standard of trust and security than one that is not broadly used or is created by a single person. Furthermore, Android Application Vulnerability Scanners are there to find easily security flaws so if you don't secure your application carefully, it will end up a target for explotation.

REFERENCES

[1] Android Architecture: https://source.android.com/devices/architecture
[2] Logcat - Android Developers https://developer.android.com/studio/command-line/logcat
[3] Android Debugging Bridge - Android Developers https://developer.android.com/studio/command-line/adb
[4] SELinux: https://source.android.com/security/app-sandbox
[5] Android Security Bulletins(Patches): https://source.android.com/security/bulletin
[6] Two-Factor Authentication: https://www.androidcentral.com/two-factor-authentication
[7] Android Sandbox: https://source.android.com/security/app-sandbox
[8] Android Webview https://developer.android.com/reference/android/webkit/WebView
[9] Diva Course videos - sinister geek https://www.youtube.com/watch?v=gHWA67mb7sU&list=PLOlyU7jql72BBJv7rXmv_ef__qE4Xyu0b
[10] Android Security https://source.android.com/security
[11] Android 12 Security Patches June 2022 https://www.xda-developers.com/june-2022-android-security-update/
[12] Android Studio Documentation for app developers - Naipeng Dong : https://developer.android.com/docs
[13] Android Request Permissions https://developer.android.com/training/permissions/requesting
[14] Understanding Android App Permissions https://guides.codepath.com/android/Understanding-App-Permissions
[15] Android Keystore System https://developer.android.com/training/articles/keystore
[16] Android KeyStore System Security Features https://android-doc.github.io/training/articles/keystore.html
[17] Android Partition system as root https://source.android.com/devices/bootloader/partitions/system-as-root
[18] MicroG GmsCore leaking https://github.com/microg/GmsCore/issues/1567
[19] Xiaomi Mi Fit SQL injection : https://github.com/satcar77/miband4
[20] Quick Android Review Kit https://github.com/linkedin/qark
[21] Quixxi Security https://quixxisecurity.com/
[b23] ndroid API Levels http://www.dre.vanderbilt.edu/~schmidt/android/android-4.0/out/target/common/docs/doc-comment-check/guide/appendix/api-levels.html