Android Operating System: Security Features, Vulnerabilities, and Protection Mechanisms

Lulwa Abdulmajeed AlJeraisy[1] and Arwa Alsultan[2]
443203455@student.ksu.edu.sa afalsultan@ksu.edu.sa
Cybersecurity Department, College of Computer and Information Sciences, King Saud University, Riyadh, Saudi Arabia

Summary
In the age of smartphones, users accomplish their daily tasks using their smartphones due to the significant growth in smartphone technology. Due to these tremendous expansions, attackers are highly motivated to penetrate numerous mobile marketplaces with their developed malicious apps. Android has the biggest proportion of the overall market share when compared to other platforms including Windows, iOS, and Blackberry. This research will discuss the Android security features, vulnerabilities and threats, in addition to some existing protection mechanisms.
Keywords:
Android architecture, Android permissions, Android security, Vulnerabilities.

1. Introduction

Android is a mobile operating system that was created with complete open-source development. The android platform should have a strong security system to ensure the security of user applications, information, and data. A strong and strict security architecture is needed to ensure security for open-source platforms such as Android [1]. The Android platform's architecture is created with multilayered security providing the needed flexibility for an open-source platform. As mobile devices are becoming more popular among users, security and privacy are also becoming a concern for smartphone users. Moreover, the rapid increase in the number of Android apps makes it challenging for app marketplaces, like Google App Store, to verify whether an application is malicious or legitimate. Additionally, since Android OS is an open-source platform and therefore is transparent to the public, vulnerabilities are easily exploited by cybercriminals. Numerous security layers of the Android operating system, including the framework layer, application layer, and even the Linux kernel layer, are susceptible to many Android vulnerabilities [2]. Benign or malicious apps are vulnerable because of unexpected design flaws or coding errors. Although Android contains powerful security features, many security threats exist, like Denial-of-Service attacks, repackaging apps, permission escalation, and unauthorized access between application services [3]. The objective of this research is focused on expanding the coverage of Android security features and threats, and in addition to the existing protection techniques.

Manuscript received November 5, 2022
Manuscript revised November 20, 2022
https://doi.org/10.22937/IJCSNS.2022.22.11.53

2. Android Security Architecture

Android incorporated many security mechanisms to protect user’s data and system’s resources, aiming to become the most secure and convenient mobile operating system available in the market [1]. To accomplish that goal, Google (Android owner) offers the following security features which will be discussed in depth throughout the research:

- Robust OS security through the Linux kernel
- User permissions
- App signing
- Sandboxed operating system
- Secure communication barriers

Before discussing the security features of Android, Fig. 1 provides a visualization of how security features are distributed throughout the Android security architecture [1]. The fundamental goal of memory space protection in the Linux kernel is to stop an operation from accessing memory without the required access permissions. Without memory protection, memory segments like code and data segments are susceptible to code injection attacks and flaws that are related to memory. By utilizing disk encryption, files are always kept on disk in an encrypted state. The application sandbox imposes restrictions on all processes that are run above the Linux kernel. The Android platform uses Linux user-based protection in the libraries to isolate application resources, unlike other operating systems in which multiple apps run with the same user permissions [3]. The whole Android operating system is built on top of the Linux kernel which is the heart of the Android architecture. It supports the running process of applications by managing all the available drivers needed during the runtime. The libraries' purpose is to support application development. It consists of the needed requirements to build an app, including the Android manifest and the source code. The application framework provides a broad abstraction for accessing the hardware and allows easier user interface management with the resources of the applications. In each layer of the architecture, there are certain security features dedicated to supporting the security needs of that specific layer. Some of

these features are going to be discussed throughout the research.

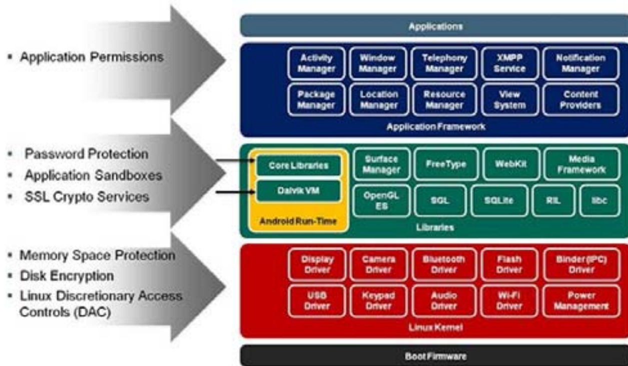


Fig. 1 Android security architecture

3. Android Security Features

The security of the Android operating system is based on the following key security features:

3.1 App Sandbox

Users can execute applications or open files in a sandbox, an isolated testing environment, without having their application, system, or platform affected. Android does this by giving every Android app a distinct user ID (UID) and running each one in a separate process. This UID is used by Android to create a kernel-level app sandbox [4]. As demonstrated in Fig. 2, an Android app's code is performed in a sandbox while executing. An app runs separately from the rest of the system and is unable to access the memory of other apps. The only method an app can access memory is using an inter-process communication mechanism which is one of the protection mechanisms provided by Android platforms. This is to prevent malicious apps from accessing other apps' data [9].

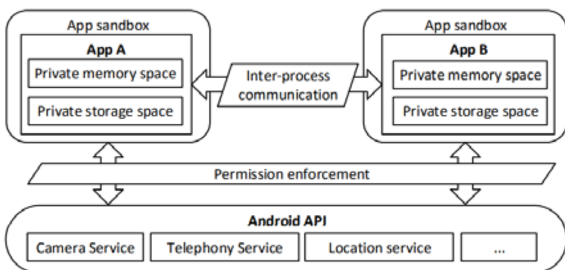


Fig. 2 Two sandboxed Android applications interacting with each other and with the Android API

3.2 Security-Enhanced Linux in Android

Android uses Security-Enhanced Linux (SELinux) as part of its security model to apply mandatory access control (MAC) on all processes, even those that have root capabilities. The SELinux feature in Android was supported by numerous businesses and organizations. With the help of SELinux, Android may more effectively secure system services, limit access to application data and system logs, and mitigate the consequences of malicious software. SELinux functions on the principle of implicit deny, which is if the access is not allowed explicitly, then deny it. It operates in two modes [10]:

- Permissive mode, logs permission denials but does not take any enforcement action.
- Enforcing mode, both logs and enforces permissions denials.

SELinux in Android operates in enforcing mode along with a security policy that works through the Android Open-Source Project (AOSP).

3.3 Trusty Trusted Execution Environment (TEE)

Trusty is a secure Operating System (OS) that runs on the same Android processor but is separated from the rest of the system. Trusty can have full access to the device's main processor and memory but is isolated completely. By running inside a TEE, malicious apps are unable to exploit the OS and cannot reach the central processing unit without direct permission [8]. Fig. 3 shows the isolation of the trusted OS and the Android OS.

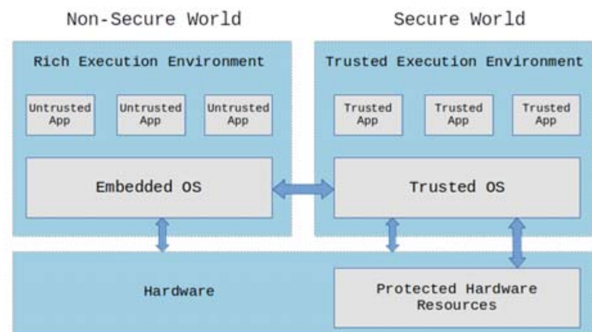


Fig. 3 Trusty Trusted Execution Environment (TEE)

3.4 App Signing

Application signing enables developers to disclose the identity of the application's developer and to update their software without having to create complex permissions. The developer must sign each Android application before it is deployed. Otherwise, the application will be rejected. The user ID associated with each program is specified in the signed application certificate in which different user IDs run different apps. Application signing protects one app from accessing another app except through inter-process communication. Application signing on Google Play serves as a link between Google's trust in the developer and the developer's trust in their application. Developers are aware that their software is delivered to the Android device unmodified [2].

3.5 Verified Boot

Verified Boot is designed to make sure that all executed code originates from a reliable source often from the original equipment manufacturer rather than from a malicious source. The verified boot uses a device mapper, which verifies the integrity of each device block. Fig. 4 shows how the device mapper uses a cryptographic hash tree to ensure the integrity of each block. Each node represents a cryptographic hash. The leaf nodes represent the hashes of the device blocks, intermediate nodes represent the hashes of their child nodes. The root node is known as the root hash, it is the sum of all hashes in the below levels. Any change made in a single device block will change the root hash value [7].

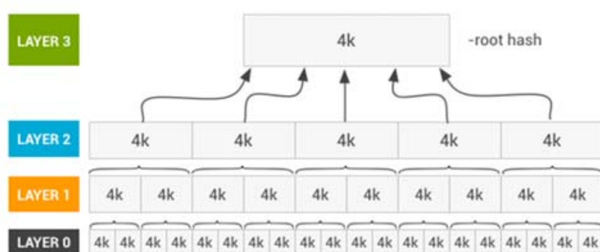


Fig. 4 Cryptographic hash tree

4. Android Security Issues

Android has a strong protection mechanism facing Android security issues, which is the application permissions system. The security of the Android operating system is designed as a permission-based approach that monitors and controls the authorization of third-party Android apps to access vital resources. This entails limiting third-party Android applications' access to crucial system resources on an Android device. Users should approve a set of permissions an application asks for, before downloading the application. This procedure aims to alert users to the potential risks associated with downloading and using the application on their device. However, even when the permissions system is carefully understood and clear, users frequently lack sufficient knowledge of the threat, which lead to them placing their trust in either the app store or the application's level of popularity, and accepting the installation without attempting to understand the developer's intentions. End users, marketers, and developers have strongly criticized this permission-based approach for the ineffective administration of permissions [3]. In this section, an overview of the Android permissions system is discussed, followed by the main security issues of Android.

4.1 Android Permissions Levels

Users can execute applications or open files in a sandbox, as the <uses-permission> tag found in the AndroidManifest.xml is used by the developer to specify the permissions. The restrictions are established at the time of the App installation. Android permissions fall under the various access levels listed in Table 1. Users, system programs, or devices are not seriously threatened by normal permissions [11]. They are automatically granted when the application is being installed and can be modified later from the settings of the application. The dangerous permissions, on the other hand, is risky as the device's sensitive APIs and private data can be accessed through these permissions [3]. For signature-based permissions, two applications need to be signed by the same certificate.

Table 1: Classification of Android permissions levels

	Levels		
	Normal Permissions	**Dangerous permissions**	**Signature-based permissions**
Description	For the user, system programs, or device, these permissions pose the minimum risk. They are automatically granted when the software is being installed. From the application settings, they can be modified later.	These permissions fall under the serious risk category because they can access the device's sensitive APIs and personal information. During the installation process, the user's permission is requested.	These permissions are only given if the requesting app's signature matches that of the application maker's certificate. During the installation process, they are implicitly granted.
Examples	ACCESS_WIFI_STATE, BLUETOOTH, SET ALARM,	LOCATION, MICROPHONE, CAMERA, CONTACTS	APP SIGNATURE, APP CERTIFICATE

	MODIFY_AUDIO_SETTINGS		

4.2 Permissions Threat Examples

Android permissions, despite being intended to safeguard users and can threaten a device's resources and data. The following are some examples of how the most popular Android permissions can be exploited [3]:

- ACCESS_WIFI_STATE

Apps can access Wi-Fi network data, including the list of configured networks and the active network, with this permission. Browser and communication apps request this permission. It can be exploited if attackers use device bugs to obtain Wi-Fi passwords and hack into the networks the user access regularly.

- MODIFY / DELETE SD CARDS

Apps can write to external storage, including SD cards, when this permission is given. This Android permission is typically needed by apps for cameras, documents, audio, and video. Attackers exploit this permission by deleting files or photos on the SD card.

- SEND_SMS

This permission is required by social media and communications applications, to grant these applications to send text messages. Attackers can use it to send messages to premium numbers, which leaves users with unexpected charges.

4.3 Android Security Levels

This section describes in detail the user and device security concerns. Google regularly releases security patches to resolve bugs and enhance device security, but malware developers continue to develop detection obfuscation mechanisms. The first step in dealing with a security bug is determining the severity of the bug and its consequences if it is exploited. The severity level assists researchers and security teams in prioritizing the issue so that the necessary bug fixes are deployed to users [6]. The severity levels along with their consequences are described in Table 2:

Table 2: Severity levels

Severity levels	Description	Consequences
Critical level	This is the most critical severity level, and it needs to be fixed immediately to keep the device secure.	• Remote boot bypass • Remote code execution • Remote data wipe • Remote DoS attacks • Unauthorized data access
High level	Bugs in this category are less critical than high-level ones but can lead to severe risk when exploited.	• Remote script execution • Bypassing the lock screen • Exploitation of cryptographic vulnerabilities • Remote ransomware attack
Medium level	Compared to critical-level and high-level defects, these bugs are less dangerous, but they still have the potential to corrupt device data.	• Bypassing the root permissions • Local script execution • Local code injection • Bypassing Wi-Fi encryption
Low level	Bugs in this category do not harm the devices much. However, they must be properly patched to secure device data.	• Removing user applications. • Random pop-up notifications for spam • Unexpected application termination

4.4 Android Attacks

The percentages of Android attacks were extremely high in 2019-2020 [1]. These vulnerabilities are dangerous to device security, and their exploitation may result in data loss, including sensitive user data. This section describes popular significant malicious attacks and their consequences. The consequences of these attacks can range from minor data loss to significant financial loss [2].

- Collusion Attack

A significant threat to Android-based devices is application collusion. App collusion occurs when two or more applications work together in some way to commit a malicious activity that would be impossible to be carried out independently. In other words, two or more applications collude to establish malicious activity. Each of the participating apps communicates with one another via legitimate communication channels to carry out the tasks assigned to each. Apps are not required to violate any security frameworks or exploit security weaknesses to carry out malicious activity [11].
Consequences: malicious activities and malware will be hidden since current anti-malware solutions are incapable of analyzing multiple apps at the same time.

- Privilege Escalation

Privilege escalation attacks target kernel-level security flaws to gain device root privileges. The attacker can

intercept publicly available device modules to gain access to vital permissions. Such attacks may emerge from unauthorized actions carried out by apps with more rights than needed, which could expose users to a great deal of sensitive data [3].
Consequence: remote access to root-level privileges might lead to full control of the Android operating system.

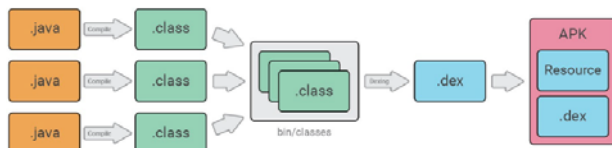
- Repackaging Apps

Using reverse engineering techniques, repackaging is carried out by injecting harmful code into the source code and decompiling/disassembling .apk files. Malicious code can be hidden by employing repackaging techniques to seem like normal software. Repackaged apps are typically corrupted variations of well-known apps. Reverse engineering is used by attackers to download popular Android apps, extract the source code, add their own malicious code, repackage the program, and then publish it. Various tools are used to obtain Java code like undx and dex2jar [5].
Consequence: attackers will release their repacked application through an unofficial market and as a result users fall prey to it.

- Denial of Service Attack

The expansion of DoS attacks has been accelerated by the increasing number of mobile devices connected to the Internet to form a large network. This attack occurs when a program fills up all the resources, such as memory, battery life, and bandwidth, preventing unauthorized users from carrying out their functions [4].
Consequence: unavailability of services to authorized users.

4.5 Real-life Attack Example



The following attack was discovered in a banking application in South Korea, where a user performs a few steps to transfer money using an Android banking application [5]. During a repackaging attack, the following steps are taken:

(1) Attacker installs the banking application.
(2) Modify and decompile the application with the attacker’s self-sign. Attackers use reverse engineering processes by decompiling the DEX file into different source codes like Java as shown in Fig. 5.
(3) Publish the application as forgery to the third-party market or Android market.
(4) Normal users install the forged banking application.
(5) Use financial transfer services.
(6) Transfer the money to the attacker’s account without awareness.

Fig. 5 Application decompiling

4.6 Repackaging Attack Protection Mechanisms

- Self-Signing Restriction

Self-signing policy prevention is one of the protection mechanisms against repackaging attacks. The simplest way to accomplish this is to switch app signing from self-signing to market signing and prohibit app distribution without the market's signature. Although this effectively eliminates repackaging attacks, it would violate Android's open policy [5].

- Code Obfuscation

Code obfuscation is a mechanism used to make it more challenging to reverse engineer source code or machine code. The default tool used for the obfuscation of the Java source code is called ProGuard. Obfuscation is supported by ProGuard by converting method names, class names, and variable names into meaningless, randomized strings [4][5].

- Code Attestation

Code is attested by the authority before performing a money transfer from the user's application to the banking server. A Trusted Platform Module (TPM) provides a stronger security mechanism supporting code attestation. TPM enables secure booting of the operating system [5].

5. Conclusion

Alongside the rapid growth of Android system usage and the number of Android applications, malicious activities are exponentially growing as well. Although there exists a strong Android security system, threats and vulnerabilities exploit the security system’s weakness in order to allow attackers access and control the system resources. In this research, key Android security features and protection mechanisms have been discussed, along with its related security issues to control, prevent, or mitigate future threats aiming for a more robust and secure system.

References

[1] Ibne, T., & Alam, L. (2016, March 17). Android Security Vulnerabilities Due to User Unawareness and Frameworks for Overcoming Those Vulnerabilities. International Journal of Computer Applications, 137(1), 14–21. https://doi.org/10.5120/ijca2016908649

[2] Ahmed, O., & Sallow, A. (2017). Android Security: A Review. Academic Journal of Nawroz University, 6(3), 135–140. https://doi.org/10.25007/ajnu.v6n3a99

[3] Alshehri, A., Hewins, A., McCulley, M., Alshahrani, H., Fu, H., & Zhu, Y. (2017). Risks behind Device Information Permissions in Android OS. Communications and Network, 09(04), 219–234. https://doi.org/10.4236/cn.2017.94016

[4] Bahman Rashidi, & Carol Fung. (2015, January 1). A Survey of Android Security Threats and Defenses. J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl., 6, 3–35. https://doi.org/10.22667/JOWUA.2015.09.31.003

[5] Jung, J. H., Kim, J. Y., Lee, H. C., & Yi, J. H. (2013, June 14). Repackaging Attack on Android Banking Applications and Its Countermeasures. Wireless Personal Communications, 73(4), 1421–1437. https://doi.org/10.1007/s11277-013-1258-x

[6] Li, J., Sun, L., Yan, Q., Li, Z., Srisa-an, W., & Ye, H. (2018, July). Significant Permission Identification for Machine-Learning-Based Android Malware Detection. IEEE Transactions on Industrial Informatics, 14(7), 3216–3225. https://doi.org/10.1109/tii.2017.2789219

[7] Johnstone, M. N., Baig, Z., Hannay, P., Carpene, C., & Feroze, M. (2016). Controlled Android Application Execution for the IoT Infrastructure. Internet of Things. IoT Infrastructures, 16–26. https://doi.org/10.1007/978-3-319-47063-4_2

[8] Sabt, M., Achemlal, M., & Bouabdallah, A. (2015, August). Trusted Execution Environment: What It is, and What It is Not. 2015 IEEE Trustcom/BigDataSE/ISPA. https://doi.org/10.1109/trustcom.2015.357

[9] Spolaor, R., Abudahi, L., Moonsamy, V., Conti, M., & Poovendran, R. (2017). No Free Charge Theorem: A Covert Channel via USB Charging Cable on Mobile Devices. Applied Cryptography and Network Security, 83–102. https://doi.org/10.1007/978-3-319-61204-1_5

[10] Building SELinux Policy |. (n.d.). Android Open Source Project. Retrieved October 22, 2022, from https://source.android.com/docs/security/features/selinux/build

[11] Karthick, S., & Binu, S. (2017, February). Android security issues and solutions. 2017 International Conference on Innovative Mechanisms for Industry Applications (ICIMIA). https://doi.org/10.1109/icimia.2017.7975551