A tale of four gates

Aldoseri, Abdulla; Oswald, David; Chiper, Robert

DOI:
10.1007/978-3-031-17146-8_12

A Tale of Four Gates

Privilege Escalation and Permission Bypasses on Android through App Components

Abdulla Aldoseri[1], David Oswald[1], and Robert Chiper[1,2]

[1] University of Birmingham, UK.
{axa1170, d.f.oswald}@bham.ac.uk
[2] robert.chiper@pm.me

Abstract. Android apps interact and exchange data with other apps through so-called app components. Previous research has shown that app components can cause application-level vulnerabilities, for example leading to data leakage across apps. Alternatively, apps can (intentionally or accidentally) expose their permissions (e.g. for camera and microphone) to other apps that lack these privileges. This causes a confused deputy situation, where a less privileged app exposes its app components, which use these permissions, to the victim app. While previous research mainly focused on these issues, less attention has been paid to how app components can affect the security and privacy guarantees of Android OS. In this paper, we demonstrate two according vulnerabilities, affecting recent Android versions. First, we show how app components can be used to leak data from and, in some cases, take full control of other Android user profiles, bypassing the dedicated lock screen. We demonstrate the impact of this vulnerability on major Android vendors (Samsung, Huawei, Google and Xiaomi). Secondly, we found that app components can be abused by spyware to access sensors like the camera and the microphone in the background up to Android 10, bypassing mitigations specifically designed to prevent this behaviour. Using a two-app setup, we find that app components can be invoked stealthily to e.g. periodically take pictures and audio recordings in the background. Finally, we present Four Gates Inspector, our open-source static analysis tool to systematically detect such issues for a large number of apps with complex codebases. Our tool successfully identified exposed components issues in 34 out 5,783 apps with average analysis runtime of 4.3 s per app and, detected both known malware samples and unknown samples downloaded from the F-Droid repository. We responsibly disclosed all vulnerabilities presented in this paper to the affected vendors, leading to several CVE records and a currently unresolved high-severity issue in Android 10 and earlier.

Keywords: Android · Application components · Multi-user · Sensors

1 Introduction

App sandboxing and isolation on widely deployed mobile operating systems like Android have brought various security benefits with it. However, due to the

need to exchange data across apps and to access system resources like sensors (camera, microphone, etc.), Android provides several externally accessible entry points into apps beyond the user interface. The four main entry points (activities, services, broadcast receivers and content providers) are called app components. Over the last few years, app components have received considerable attention from security researchers [23, 16, 37]. Numerous vulnerabilities have been found that led to, among others, privileges escalation and side-channel leakage of user data [38, 16]. However, less attention has been paid to how these components interact with Android-wide restrictions, e.g., the separation of user-profiles and the permission system. In this paper, we address this issue and show how exposed app components can be used in unintended ways to break Android's Multi-User (MU) feature, (mis)use permissions held by other apps, and conceive a construction that allows accessing the camera and microphone in the background on recent Android versions. Subsequently, we introduce Four Gates Inspector, an open-source static analysis tool that can aid developers in detecting and preventing certain issues discovered in this paper.

1.1 Contributions

In this paper, we systematically analyse exposed app components across system restrictions and multiple user profiles on the same device. Based on our analysis, we discover several vulnerabilities impacting user privacy and security. We first show that users with the `INTERACT_ACROSS_USERS` or `ACCESS_CONTENT_PROVIDER_EXTERNALLY` permissions can invoke app components belonging to other user profiles. As the `adb shell` user has this permission by default, an attacker with either physical or remote access via WebUSB can misuse those interfaces to install apps into another user's profile, grant arbitrary permissions, and then exfiltrate the data through app components. Notably, the `shell` user does not have read/write access to other user profiles through standard means (e.g., using filesystem commands like `ls`, `cd`). The attack bypasses the dedicated lockscreen of the target user profile.

Secondly, we show that app components allow adversaries to construct stealthy spyware that can access sensors like the camera and microphone in the background. This is achieved by installing two apps: one app exposes access to the sensors through app components, and a second (unprivileged) app repeatedly invokes this functionality in the background. This issue bypasses countermeasures against background spyware for recent Android versions up to Android 10, thus affecting all Android devices in use at the time of reporting (Sep, 2020) [31].

To mitigate the issues presented in this paper, we propose Four Gates Inspector, a static analysis tool to detect the usage of a given class or API (e.g., camera or microphone) by tracing the invocation of methods of each app component using graph-based analysis. Additionally, the tool aids in detecting confused deputy issues in exposed app components [22]. Our main contributions are:

- We perform a systematic and empirical analysis of app components, including communication across user profiles, visibility of the user, and restrictions.

- We analyse the main four implementations of the Android multi-user feature (Samsung secure folder, Huawei private space, Xiaomi second space, Google multi-user) and show how to bypass Android lock screen protection in them, giving full access to an adversary through app components.
- We show how spyware can use app components to stealthily access camera and microphone in the background, breaking the OS countermeasures against such techniques up to Android 10.
- We present Four Gates Inspector, our open-source static analysis tool to detect the use of specific APIs (e.g., sensor access) in app component handlers. We show that Four Gates Inspector can not only detect known samples of background spyware, but also identified confused deputy issues in 34 (benign) apps (out of a sample of 5,783) downloaded from F-Droid [18] with an average runtime of 4.3 s per app.

1.2 Responsible disclosure

The vulnerabilities described in this paper have been responsibly disclosed through the proper channels. Samsung secure folder's issue was reported on August 18, 2020, assigned moderate severity, and can be tracked via CVE-2020-26606. The vulnerability of Huawei's private space was reported on August 18, 2020, assigned moderate severity, and can be tracked via CVE-2020-9119. We reported Xiaomi's second space issue on August 28, 2020. However, Xiaomi informed us that this issue had been reported in parallel by a third party and was fixed on August 31, 2020. Finally, we reported Google's multi-user feature issue on Sep 17, 2020. In contrast to the other vendors, Google considers this as intended behaviour as does not intend to deploy a fix.

The stealthy background access to camera and microphone issue was reported to Google on Sep 18, 2020, and assigned a high severity. Google then confirmed that the issue has been fixed in Android 11. Google did not assign a CVE to this issue, as our report coincided with the release of Android 11 and thus did not affect the latest Android OS. The issue can be tracked via id 175232797 on Google issue tracker We further explored the respective changes in Android 11 and found that Google re-designed the permission system to specifically mitigate this issue [9].

To ensure the reproducibility of our work and to provide the community with a relevant sample of vulnerable apps for evaluating future attacks and defences, we provide our code, demo videos, including Four Gates Inspector at `https://akaldoseri.github.io/a-tale-of-four-gates/` and `a-tale-of-four-gates` repository at `github.com`.

2 Background

We first give an overview of app components and their security aspects.

App components Mobile apps are isolated and sandboxed by Android OS in a per-app Virtual Machine (VM). For app communication, Android offers mechanisms to share data between apps. Among them is app components where they serve as entry points allowing communication between apps. They are widely used in practice. Thus, we focus on this aspect. There are four types of app components: *activity*, *service*, *broadcast receiver* and *content provider*. An *activity* consist of a user interface and an executable code section. *Services* run in the background without user interface/interaction. *Broadcast receivers* respond, once registered, to broadcasted Android Intents. *Content providers* abstract the interaction with app data, e.g., files or SQLite databases [5].

App components are accessible to other apps if their respective `exported` attribute in the manifest is set. They can be restricted by assigning them app-specific (custom) permissions which provide a protection level from low (*normal*) to higher levels (*dangerous*, *signature*, and *signatureOrSystem*) [8]

Multi-user feature In addition to app isolation, Android also provides a multi-user feature that allows to set up several isolated user profiles on a single device. Each profile has a workplace to store data and install apps [27]. To prevent users (including the `shell` user) from accessing each other data, Users' data is stored in a separate virtual area in the internal storage [4]. Additionally, a user can lock their profile via Android Gatekeeper (e.g., through their PIN or fingerprint) [3]. Use cases for this include a device shared by family members or an on-call team.

3 Related work

Privileges escalation on custom permissions Custom permissions have a unique name and protection level to ensure access protection for their allocated resources. Several issues have been reported in custom permissions related to OS and app updates: Xing et al. showed that one can claim ownership of resources (e.g., permissions, package names, etc.) by defining them in an app in an old Android OS before they are actually introduced in an OS update [34]. (e.g., a malicious app can obtain the system permission `ADD_VOICEMAIL`). Tuncay et al. showed that an adversary can obtain signature permissions from third-party apps without signature matching. For this, the adversary first installs two apps, where one defines the target permission and the other grants it. By uninstalling the first app and installing the victim app, the second app gains access to the victim app using its target permissions, regardless of signature mismatches. This is because Android OS does not revoke granted permissions by default [33]. Considering the properties of Android that caused these issues, Rui Li et al. proposed a fuzzer to detect such problems by evaluating custom permissions in randomly created apps installed on devices across different test cases, including system update and app update. Their work demonstrates several issues that allow adversaries to obtain sensitive permissions without user consent [25].

Exposed app components Confused deputy attacks [22] are vulnerabilities in which an (unprivileged) attacker misuses permissions granted to a higher

privileged component due to missing checks in communication interfaces. In the context of mobile devices, it has been repeatedly shown that this problem can manifest in apps [20, 28, 33, 15].

In most cases, the underlying reason was the exposure of app components to all apps on the device by setting their `exported` flag. For instance, as a consequence, a content provider that manages private app data may expose full read/write access to this data if no restrictions are imposed [38]. Therefore, researcher have focused on developing tools and methods to detect such issues. Zhou and Jiang systemically analysed 62,519 apps w.r.t. two different impacts: data leakage and denial of (some) Android services. Their analysis showed that 1,279 (2.0%) and 871 (1.4%) apps were vulnerable to those issues, respectively [38]. Heuser et al. propose DroidAuditor, an Android module that observes apps behaviour at runtime and generates a graph-based representation of access to sensitive resources (e.g., camera, SMS, etc.) to inspect collusion attacks and confused deputy attacks [23]. Similarly, Yang Xu et al. developed a framework to detect permission-based issues by collecting runtime app states and applying policies and capability-based access control to mitigate these issues at runtime [35]. Bugiel et al. implemented a detection tool that monitors IPC communication at runtime and uses heuristics as well as detection rules [14]. Reardon et al. proposes a detection tool for covert channels and side channels in apps by monitoring an app's runtime behaviour and network traffic, whereas the interaction with the apps was automated with a user interface fuzzer [28]. Felt et al. developed an IPC inspector tool that revokes permissions temporarily when apps communicate with each other. The revocation process keeps only the commonly granted permissions between the communicating apps [20]. As a limitation, these solutions require changes on the OS level and access to high-privilege services. Also, their dynamic analysis nature may leave some code paths unexplored if only triggered by specific inputs, e.g., on a login screen [28].

Alternatively, other researchers have proposed approaches based on static analysis. CHEX is a taint-based method to detect leakage in exposed components [26]. AppSealer follows CHEX' approach, which is based on TaintDroid [36, 17]. AppSealer additionally introduces a patch code generator to patch the detected issues. Zhong et al. utilise tainting to detect these issues between selected pairs of applications by comparing their permission and performing inter-application control flow analysis [37]. Finally, Elsabagh et al. propose FIRMSCOPE, a static analysis tool based on practical context-sensitive, flow-sensitive, field-sensitive, and partially object-sensitive taint analysis [16]. It detected several privilege escalation vulnerabilities in system apps, including code execution using exposed components. FIRMSCOPE achieves a better performance than its predecessors FlowDROID [13], AmandDroid, and Droidsafe. Overall, methods based on taint analysis are limited by their high performance costs, which make them less practical [35], and over-tainting problems that may lead to false positives.

To overcome these limitations, in Section 6, we propose a static analysis tool that detects (i) confused deputy issues and the potential bypass of system restrictions and constraints (e.g., using the camera in the background) and (ii)

exposure of certain libraries (e.g., tracking and scoped storage). For (i), we noticed that [20] does not consider two apps that expose the use of the same permission (e.g., camera) and communicate with each other. However, in Section 5, we show that this can bypass Android OS restriction to access the camera in the background. For (ii), most previously proposed solutions rely on permission settings. Recently, noticeable operations like scoped storage [7] introduced in Android allow access to the internal storage for read/write without storage permission, which might not be detectable by existing tools. Finally, given that taint-based approaches are computationally costly, we decided to investigate the alternative approach of statically analysing the execution trace of app components in a graph-based representation and detecting issues on the level of Smali code.

Compromising user privacy Starting from Android 9, Android prevents background services from accessing the camera and the microphone, even if they have the necessary permissions. Apps with foreground services can still use the camera and microphone without the app being in focus, but only if they display a persistent notification to users [2]. Sutter and Tellenbach discovered a race condition in this functionality: quickly hiding the foreground service's notifications allowed them to create spyware that uses the camera and microphone without showing any visible notification [32]. In Android 11, the camera and microphone can be used only while the app is active and in use (*i.e.*, in the foreground) [11]. In our work, we bypass the restriction that prevents access to camera and microphone while the app is in the background. We achieve this by accessing these sensors via exposed components from another app, which the activity manager considers to have foreground visibility

Containers and multi-user feature The Android multi-user feature has received considerable attention from the research community: Ratazzi et al. performed a systematic security evaluation, considering a threat model where multiple users share the same device. They discovered several vulnerabilities, e.g., that secondary users can access the owner's services like VPN, network, backup and reset settings because they are exported publicly. OEM-specific functionality derived from the multi-user feature has suffered from several issues in the past: the security space service that manages Xiaomi's second space allowed an adversary to switch to the second space without user authentication [19]. This was because the service was publicly accessible, and an adversary could start it with a crafted intent to bypass the authentication process. Kanonov and Wool reported several issues in Samsung Knox containers, including a man-in-the-middle vulnerability in the VPN service and data leakage from the Knox clipboard [24]. We demonstrate an issue that allows extracting data from either multi-user or OEM-specific spaces, bypassing the dedicated lock through the shell user permissions.

4 Analysis of app components across user profiles

Starting from Android 5, Android supports a multi-user feature that allows adding additional users to the device so that a device can be shared by multiple people [4].

Regarding the security guarantees provided by the multi-user functionality, the relevant documentation states that "each user gets a workspace to install and place apps" and "No user has access to the app data of another user" [4]. Technically, as discussed in Section 2, each user's data is stored in a separate virtual area to prevent accessing the data of other users. Users can lock their profile via Android Gatekeeper [3]. However, apps can interact with each other using app components. Thus, the question arises if it is (i) possible to invoke app components of an app in one user profile from another profile, and (ii) if this can be used to bypass the user-specific lock screen.

Accessing app components with different users We experimentally investigated interactions between an app residing in one profile and another one in another profile. We started by creating two users, "adversary" and "victim", and two apps in each profile: A callee app with four exposed app components (activity, service, background, and receiver), and a caller app to invoke the victim app components. Our initial analysis showed that apps can only interact with each other if they are in the same profile. Debugging the caller to send messages via `adb` using the activity manager `am`, and `content` tools showed that messages target apps in the same profile by default. Yet, we found that both tools accept a `--user` flag to specify the user ID of the target profile unlike their equivalent Java API. Specifying a user ID (obtained via the package manager `pm`) in the messages results in a permission denied exception from the `UserController` class. We thus confirmed that apps cannot communicate across profiles by default.

Underlying permission After analysing the exception, we found that `UserController` class manages multi-user functionality for `am` and, a user can interact with app components from other profiles if it has `INTERACT_ACROSS_USERS_FULL`, `INTERACT_ACROSS_USERS` permissions for Android intents, and `ACCESS_CONTENT_PROVIDER_EXTERNALLY` permission for content providers.

These permissions are granted only for system apps with signature protection level (*i.e.*, apps signed with the system image certificate) [6]. An accessible user is the `shell` user (implemented by `com.android.shell`). We analysed its respective APK file and found that the shell package has all these permissions. Utilising `dumpsys`, we confirmed that the `shell` user is effectively granted these permissions.

By default, the `shell` user does not have read/write access to other users' profiles. Using commands like `ls` or `cd` to access them leads to insufficient permission errors. Therefore, we focus on exploiting these permissions. We sent both an intent and a content query using the shell, specifying the victim's user id, and noticed that sending an intent to an activity does trigger the lockscreen protection (Gatekeeper). However, for services, broadcast receivers and content providers, this is not the case, as they do not have a visible user interface. We thus could successfully bypass the Gatekeeper protection and interact with apps from a different profile. (e.g., a secondary user able to extract information from Owner user). Yet, practically exploiting this either requires a vulnerable app in the target profile, or the ability to install specific apps (e.g., an app that exposes user information through an app component) into the victim profile.

Threat model Ratazzi et al. consider a threat model based on physical access to the device where a single device is shared by multiple users to show data leakage issues in the multi-user feature [27]. Such a model applies to our issue. However, we also consider a more generic threat model, where a victim uses a service that requires access to ADB (e.g., Vysor, GenyMobile, MirrorGo, ApowerMirror). These services are widely used for screen mirroring, operate on desktops or browsers via WebUSB, and have over 12M users in total. Because ADB gives the service access to potentially sensitive data in the user profile, we further assume that the user has created a second profile to limit the information accessible via ADB. Clearly, this threat model may be seen as specific and may require additional social engineering or similar circumstances. It does not require knowledge of PIN or fingerprint of the target profile or rooting the device and is thus applicable to Android devices where rooting is detectable, e.g., through a warranty bit [29] or where PIN/fingerprint are protected by secure hardware.

Data extraction across user profiles To demonstrate a proof-of-concept of data extraction from another user profile (e.g., private profile or secure space), we first observed that the package manager system binary `pm` also accepts a `--user` flag, similar to `am` and `content`. `pm` exposes a range of app management operations, including app installation and granting of arbitrary permissions. Thus, a combination of `am`, `pm`, and `content` allows to fully bypass the isolation between user profiles for a `shell` adversary. Concretely, the attack proceeds as follows:

- The victim visits an adversary-controlled website with ADB support (e.g., disguised as a screen mirroring service) and connects their device to it via WebUSB.
- The adversary's website silently installs a malicious app into the victim's other user profile (e.g., one created to separate sensitive data from a profile used for screen mirroring) through `pm`.

```
$ pm install com.app.malicious.apk --user 2
```

- Using `pm`, the adversary grants the malicious app read access to the victim user space's internal storage (e.g., `READ_EXTERNAL_STORAGE`) (or other permissions).

```
$ pm grant com.app.malicious READ_EXTERNAL_STORAGE --user 2
```

- Finally, the adversary use the `shell` user to either send an intent or a content query to the malicious app to communicate with its app's components (service, broadcast receiver, or content provider) and instruct it to upload data (e.g., pictures and content of the internal storage) to a server under attacker control.

```
$ content query --uri content://com.app.malicious/cp --user 2
```

- To hide the attack's traces, the malicious app can finally be removed via `pm`.

Alternatively, the above attack can also be carried out by an adversary with physical access to the device and a user profile, e.g., an abusive partner or family member of the victim. Apart from that, an adversary can also perform other actions using the system's Media Content Provider, which abstracts the

interaction with a user's gallery. This includes: (i) reading all images from the victim user profile, (ii) placing own images into the victim user profile, possibly with incriminating content, and (iii) Denial-of-service (or ransomware-like) attack by deleting data (or threatening to do so) inside the victim user profile.

Vendor-specific multi-user implementations The above issue was demonstrated for the multi-user feature of Google devices. Other vendors chose to apply modifications to this functionality and refer to it with their names: Samsung's secure folder, Huawei's private space, and Xiaomi's second space, which all provide similar features to the default Android multi-user mode. However, some of these variants have been enhanced with protections to prevent access in case of rooting the device e.g., through Samsung Knox [30, 29] or while the device is in debug mode for Huawei.

We thus analysed to what extent those implementations suffer from the same issues as the default multi-user function. For all tested variants, an `adb` shell provided a user with the required `INTERACT_ACROSS_USERS` or `ACCESS_CONTENT_PROVIDER_EXTERNALLY` permission. We further found that, depending on the vendor, some restrictions were imposed on the `pm`, `am`, and `content` binaries. Table 1 in Appendix A summarises our findings.

Apart from Samsung secure folder, all implementations allowed installing apps in the victim profile and were thus vulnerable to the above attack, and also exposed full access to the gallery through the Media Content Provider. For Samsung, we compared to version of secure folder (1.2.32 for older devices below Android 9 and 1.4.06 for new devices). In both cases, installing apps was prevented, however, the Media Content Provider exposed full image data (for 1.2.32) or at least metadata (for 1.4.06), e.g., timestamps and geolocation.

Samsung and Huawei assessed the respective vulnerability as "moderate". For those vendors, the issue can be tracked via CVE-2020-26606 and CVE-2020-9119, respectively. Xiaomi was already aware of this issue at the time of our report and rolled out a fix shortly thereafter. Google regarded the issue as intended behaviour and thus does not plan to release mitigations.

5 Analysis of sensor background access

Based on Section 2, we determined that app components are capable to be started by any other app with appropriate permissions. Also, app components like services, broadcast receivers and content providers are invisible to the user (no notifications), but can nevertheless access certain Android APIs, including sensors like camera and microphone (if the app has the required permissions). Since Android 9, the OS prevents apps to access those sensors in the background, even if the app has the required sensor access permissions [2] to ensure user privacy. This restriction applies unless a foreground service displays a persistent notification to the user indicating its activity [11]. This raises the question of how Android distinguishes between "foreground" and "background" at a technical level, and especially how invocations of app components like content providers and services are handled w.r.t. to this aspect.

Importance of app components To answer this question, we developed a test app with four app components (activity, service, broadcast receiver, and content provider). Each component then dumps the current "importance" of the app. This importance refers to a numerical value that precisely specifies the visibility and foreground/background state of the app [1]. Generally, the foreground is given an importance of 100, while larger values indicated states increasingly in the background.

We noticed that the app has an importance of 100 (foreground) when it is visible to the user, regardless of any app components used. When the app is minimised and stopped, the observed importances range from 125 (foreground service) over 200 (visible) to 300 (service). Using the activity manager's `start-foreground-service` option, background services can be run as foreground services and be assigned their respective importance, even if they are not developed as foreground services [11]. This highlights that "background" in Android OS does not refer to a single state, but actually a range of importances. As Android prevents access to the camera and microphone in the "background", it is important to determine for which importance this block is actually active.

To determine this, we developed two apps: the first app has three app components (service, broadcast receiver and content provider). We exclude the activity because it always runs in the foreground and thus visible to the user. We granted access to the microphone and the camera to the app and intentionally exposed these functions through all three app components. The second app then invokes the app components of the first app. We configured the intent for accessing the broadcast receiver and the service with `FLAG_INCLUDE_STOPPED_PACKAGES`, allowing it to invoke stopped apps. Crucially, we found that *all three app components were capable to access the camera with the first app in the background*, while the microphone was accessible from the service and content provider. No notification is shown to the user, making exploitation of this issue stealthy. Yet, in the initial proof-of-concept, the second (invoking) app was in the foreground, making practical exploitation obvious. To avoid this, we next consider how to invoke the respective app components from a background app.

5.1 Stealthy background spyware

Android provides several services (e.g., Timer, JobSchedule, AlarmManage and Runnable) for an app to persist in the background after it has been closed. While direct access to the camera and microphone is blocked in the context of these services by Android to prevent background spyware, app components can be invoked using these methods. As the timer class provides a minimum invocation interval of 1 s, it represents the best choice for an adversary to frequently capture camera images and audio through a second app's components.

Combining the caller app running in the background with a callee app that exposes sensor access through app components, stealthy, persistent spyware can be devised. Concretely, the attack proceeds as follows:

– The adversary installs (or tricks the user into installing) a *sensor app* that exposes a suitable app components (service, broadcast receiver, or content

provider). This app must be granted sufficient permission to access the respective device sensors (*i.e.*, camera and microphone).
- The adversary then installs (or tricks the user into installing) the second *spyware service app* (1). This app has a single background service that starts a timer. This timer periodically communicates with the app component of the sensor app to record images and audio (2), and e.g., then upload them to a remote server under adversary control. The spyware app does not require any permissions and its service can run in the background (3).

The spyware app periodically communicates with the app component of the sensor app in the background. This will bring the sensor app temporarily to importances of 125, 200, and 300 (foreground service, visible, service), enabling access to all device sensors. While the pair of our attacker apps are running, there are no user-visible indications or notifications. Because the service app does not consume substantial device resources, after 3–5 s of running, Android will white-list it and consider it a cache process, which will not be terminated. The cache process can hence remain active for a long time—in our tests, it ran continuously for more than two hours until we force stopped it. The attack works regardless of using other apps, using the camera, or locking the phone.

Reviewing the source code [21] of Android 10 shows that the camera and microphone rely on the activity manager to determine if the calling app is active (not idle or in the background) by checking its proc state. The proc state value is a numeric value convertible to an importance. The activity manager implementation translates the observed importance of 125, 200, and 300 into the proc states of foreground service, foreground, and service, which are all considered not to be in the background and thus allow bypassing sensor access restrictions.

Threat model We consider the "two-app setup" threat model widely used in the literature [20, 28, 33, 15]. We assume an adversary that can install two apps on the target device (instead of one for "classical" spyware). This adversary model is realistic for spyware, where the attacker (e.g., an abusive partner) might have temporary access to the device or might be able to social-engineer a victim into installing apps disguised as harmless software (e.g., games or other apps by the same developer). Additionally, we consider the threat model of [38], where a benign app exposes access to sensors via exposed components: a malicious app can scan for such vulnerable app components and thus "inherit" their permissions. According to our experiments, the issue affects all devices running Android 10 and earlier. It does not replicate on Android 11 due to changes in the permission system as discussed in Section 7. Our attack does not require root or `adb shell` access and hence also applies to devices where no root exploit exists or where rooting leaves traces. As we detail further in Section 6, we found instances of this issue in real-world apps.

6 Evaluation

In this section, we evaluate the discovered issues to assess their impact and provide a detection mechanism for security researchers and mobile developers.

First, the multi-user issues only affect OEM devices and are somewhat limited. Therefore, debugging the device using the *pm* and *am* binaries as discussed in Section 4 is sufficient for detection. A limitation of this approach might be that some devices do not have developer options or the shell user is inaccessible.

Conversely, the background spyware issue affects mobile apps and requires analysing the app to detect it. Therefore, we designed Four Gates Inspector to systematically discover corresponding issues and resolve limitations of existing solutions. Security researchers and mobile developers can use our tool to detect possible malicious apps. This is especially relevant when vendors pre-install third-party apps as part of system images. In the following, we discuss the design, implementation and practical results obtained with Four Gates Inspector.

6.1 Four Gates Inspector

As discussed in Section 3, runtime analysis tools previously proposed in the literature might not reach code paths that require specific input, e.g., login screens [28] and require changes to the OS, which makes them less practical. Additionally, taint-based solutions suffer from non-negligible computational cost and over-tainting that may lead to false positives [35]. To overcome this limitation, we investigate the potential of developing a static analysis tool that aids in the detection these issues. We designed Four Gates Inspector to statically analyse apps and detect confused deputy issues based on the usage of given classes and methods. Our tool is based on analysing the execution trace of the decompiled Smali code of app components. Four Gates Inspector offers a filter list that limits the scope of the analysis (e.g., camera usage, storage leakage, GPS). It provides fine-grained control over the analysis scope on the class level, compared to the permission level used by Bugiel et al. 's framework [14]. This allows us to trace issues related to non-permission classes, e.g., scoped storage in Android 11 [7].

Design and Implementation Four Gates Inspector focuses on the execution trace of method invocation of app components, in contrast to taint-based solutions that trace the input and may introduce over-tainting [35]. We thus avoid false positives (e.g., due to usage of sensors unrelated to app components) and can also detect attempts to hide sensor use (e.g., through nested calls). Our tool is implemented in Python and requires `apktool` [12] to decompile mobile apps and `untangle` to parse XML files. Four Gates Inspector consist of four main modules: *Smali Handler*, *Manifest handler*, *APK handler* and *Component inspector*.

Overall, Four Gates Inspector implements the basic flow shown in Figure 1 to analyse an app. First, the *Component inspector*, which controls the flow of the tool, starts by loading the application file (APK) and the filter list (1). The *APK handler* unpacks the app to extract the Smali code and Android manifest (2). The *Manifest handler* parses the app's Android manifest to detect app components and the use of specific classes and APIs (e.g., camera, microphone or storage) (3). To this end, the *Smali handler* initiates static analysis. It receives the component name from the Android manifest, builds a stack trace of all invoked methods and

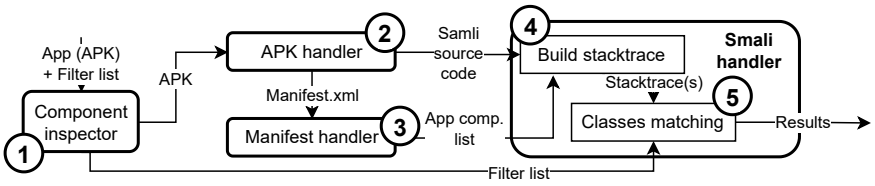

Fig. 1. Flow of Four Gates Inspector for analysis of app components.

their classes for each component by parsing their Smali source code, and explores the stack trace for the usage of filter list (4).

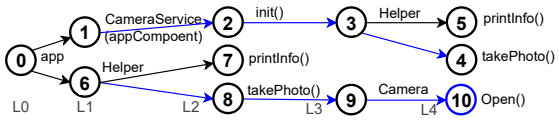

Fig. 2. Stack trace tree generated by Four Gates Inspector.

The stack trace (Figure 2) is a tree-structured graph `G(N,V)`: each app component has its `G(N,V)` tree, where `N` is a set of nodes that represent the invoked classes/methods in the execution trace, and `V` is a set of vertices that represent the invocation sequence between nodes. Figure 2 shows an example of the generated stack trace tree of an app that uses the camera through a `CameraService` component via a custom `Helper` class. The graph starts with the single root node `n0` for the app package class. Node `n1` at level 1 represents an app component class. The *Smali handler* starts by identifying method and class invocation within `n1`. The top-level invoked methods in `n1` (e.g., constructor, etc.) are added as children to `n1`. Then, each method at level 2 is analyzed to find further invocations to be added to the tree at level 3. The level 3 nodes might have an invocation to external classes and methods. These external invocations are added as child nodes to the root node `n0`. The process is repeated, starting with the Smali handler, for the new level 1 nodes, until all code has been traced. Once the stack trace is complete, the Smali handler performs tree traversal for each stack trace to detect the use of certain classes/methods. This is repeated for each component. Finally, Four Gates Inspector produces a summary report, containing detailed information about the inspected issues, class usage, stack traces, (custom) permissions and exported flags for each usage of the filtered APIs (5). Each stack trace of an app component is generated independently and reported to allow partial processing of apps, which avoids exhausting the memory for mass scans. Additionally, the graph approach includes measures to prevent infinite loops or recursions by ensuring that nodes at level 1 have unique values to avoid exploring the same classes unless new methods are invoked.

6.2 Real-world tests

We evaluated the correctness and effectiveness of the tool using two test samples:

Known samples test: We analysed two known samples for use of the camera in the background. The first sample is the proof-of-concept for CVE-2019-2219, utilizing a race condition in a foreground service to use the camera in the background [32]. The second sample is our own proof-of-concept from Section 5.

Unknown samples test: To detect unknown instances, we downloaded 6,687 apps from F-Droid, a repository of free and open-source apps [18]. The apps are from multiple categories including navigation, development, graphics and system apps. We also considered multiple versions of an app to detect issues that were fixed (or introduced) with new versions. After eliminating duplicates, this yielded 5,783 apps. We analysed these apps with Four Gates Inspector, filtering for the camera, microphone, hardware sensors (accelerometer/gyro) and GPS use. This was followed by a manual analysis of the detected issues to confirm they are true positives. We did not use commercial apps, e.g., from Google Play, to avoid legal issues with terms and conditions that prohibit reverse engineering—this decision was taken because manual screening of the terms of thousands of apps is infeasible.

6.3 Evaluation and results

In the known sample test, as expected, Four Gates Inspector successfully detected both spyware samples and indicated the exact instruction where camera and microphone are used. In the unknown sample test, the tool successfully generated reports for all 5,783 apps. Of these, 151 apps used sensors. Filtering for exported components apps yielded 34 apps with 43 components using sensors. We evaluated the correctness of the tool by manually analysing the Smali code of all those 34 cases of sensor use. We found that all the 43 cases were true positives and Four Gates Inspector correctly identified sensor usages.

In terms of performance overhead, we measured the execution time for analysis of each app under Ubuntu 16 on an i7 with 16 GB RAM. We calculated the average runtime of over 5,783 apps to be 4.3 s, which is reasonable for mass analysis, especially considering that our tool can be run in parallel. Comparing our tool to existing solutions, Four Gates Inspector is faster than other tools: AppSealer and FIRMSCOPE report 1–3 min and 7 min on average, respectively, to process an app. Note that the authors of AppSealer used a machine similar to ours (Intel i7 with 8 GB RAM), while FIRMSCOPE ran on a high-performance server (Intel Xeon 40-core processor with 150 GB RAM).

Remarkably, Four Gates Inspector found two instances of confused deputy issues in the tested apps, highlighting the practical relevance of our approach: `com.commonslab.commonslab` and `org.wikimedia.commons.wikimedia` , both used to access Wikimedia Commons, expose the audio recording permission to any unprivileged app on the same device.

We did not find instances of clearly malicious spyware in our sample collection, indicating that this type of malware is not common on open platforms like F-Droid. Comparing our results to Zhou and Jiang's study from 2012 [38], we

observe a clear improvement in the adoption of proper security practices: most apps disabled the `exported` flag for their app components. Additionally, our tool can correctly distinguish between vulnerable and secure versions of apps. For instance, only version 9.1 of `net.majorkernelpanic.spydroid` had a component using a sensor, but not earlier and the later versions.

6.4 Limitations

Four Gates Inspector has certain limitations: while it handles inner classes, interfaces, extended classes, and case-sensitive naming (tested by considering different versions of the apps). However, the tool cannot detect overridden methods unless their invocation can be traced (e.g., within the app component). This is because Smali code does not label them, which makes detection difficult especially for event-based (e.g., `onStopped`, `onResume`, etc.) and native classes not shipped as part of the app package. Secondly, our tool does not detect runtime-evaluated code, which includes dynamically-registered broadcast receivers, reflection, `eval()`, and similar. To overcome this issue, the Smali handler would need to be changed to semantically analyse such instructions and then process them using the usual flow. We leave this for future work and welcome improvements from the community.

7 Discussion and mitigation

Multi-user feature The vulnerabilities in multi-user functionalities (Section 4) allow an adversary with *shell* access (remotely via WebUSB or physically) to the device to bypass the dedicated Gatekeeper protection and extract data from other user profiles. We conclude that this issue is caused by granting the `INTERACT_ACROSS_USERS` and `ACCESS_CONTENT_PROVIDER_EXTERNALLY` permissions to the `shell` user. In addition, services, broadcast receivers and content providers do not have a user interface, hence Gatekeeper protection is not being invoked when such app components are accessed across profiles. We suggest the following mitigations:

Removing the permission from the shell user: Not granting these permissions to the `shell` user would resolve the issue for both the default Android implementation and vendor-specific variants.

Sanitise the user flag of system binaries: Alternatively, system binaries that allow access across profiles (`am`, `pm` and `content`) could blacklist user IDs of other profiles. This is especially applicable to vendor-specific implementations, as they provide a single secure space with fixed user ID. In contrast, Android's multi-user feature allows creating several users with different user IDs. Instead of blacklisting certain IDs, a whitelisting approach might be applicable.

Samsung seems applied the second approach, blocking the user ID of their secure spaces for `am`, `pm`, and `content`. While Huawei's patch include verifying the password of private space when the developer mode is enabled and inform the users about the developer mode's risks when the developed mode is enabled and entered. The mitigations taken by Xiaomi are unclear, as a parallel disclosure led them to not engage further during the patching process. As mentioned, Google

regards the problem as intended behaviour and does not plan to roll out a fix. Thus, the issue does not apply to the latest version Android 11. We note that users of affected devices (e.g., Google and Android One like Xiaomi and Nokia) might not be aware of the threat model assumed by Google internally and use multi-user with wrong assumptions about its security guarantees.

Background access to restricted sensors The root causes of this issue is the definition of "background" in Android. We suggest the following mitigations:

Restrict sensor access from app components: Background services, broadcast receivers, and content providers should generally not have access to restricted sensors when the app is invisible. However, applying such a change old existing Android version might obviously break app functionality.

Restrict sensors access when an app is not in use: This would require redesigning the camera and microphone permissions to ensure that sensors are only accessible when the app is visible in the foreground and in use. Based on our understanding and testing of the behaviour of Android 11, we noticed that it relies on the "AppOpsService" [10] to mitigate this issue. The service tracks the app's proc state (importance) using mainly two modes, `MODE_ALLOWED` to allow and `MODE_IGNORED` to suppress access. The service receives the current proc state of an app and capability updates from ActivityManagerService, indicating "while in use" permissions. This restricts sensor access as follows: If the app is in the foreground, the mode is `MODE_ALLOWED` and behaves normally. When the app goes into the background, this changes to `MODE_IGNORED`, preventing sensor access. Note that `MODE_IGNORED` does not revoke the granted permission.

Google assigned a high severity to our issue and consider it fixed in Android 11 only since our report of the issue coincided with the release of Android 11 (Sept 2020). However, in practice, the privacy impact on users of older versions might be substantial: Many—especially non tech-savvy—users might not choose or be able to upgrade their devices to Android 11. As of Jan 2022 (two years after the Android 11 release), the majority of Android users (64.63%) still use Android 10 and below [31], without any easily available mitigations.

8 Conclusion

In this paper, we demonstrated how app components on Android can be abused to break some of the system's security and privacy guarantees. We show that they can be accessed across user profile boundaries, leading to data exfiltration for a range of vendors. We demonstrate how spyware can bypass Android's mitigations against background access to the camera and microphone by splitting the process into two apps that communicate over certain app components. Finally, we present our tool Four Gates Inspector that can automatically detect such issues through static analysis. Using the tool, we e.g., found issues in apps that expose the microphone to any unprivileged app on the same device.

Acknowledgement

We thank Ali Darwish, ElMuthana Mohamed, Hamad Salmeen, Abdulla Subah and Zhuang Xu for participating in mobile testing. This research was partially funded by the Engineering and Physical Sciences Research Council (EPSRC) under grants EP/R012598/1 and EP/V000454/1. Abdulla Aldoseri is supported by a stipend from the University of Bahrain

Bibliography

[1] Android: ActivityManager.RunningAppProcessInfo. https://developer.android.com/reference/android/app/ActivityManager.RunningAppProcessInfo (2020), accessed on 09/17/2020

[2] Android: Behavior changes: all apps. https://developer.android.com/about/versions/pie/android-9.0-changes-all (2020), accessed on 09/13/2020

[3] Android: Gatekeeper. https://source.android.com/security/authentication/gatekeeper (2020), (Accessed on 11/01/2020)

[4] Android: Supporting multiple users. https://source.android.com/devices/tech/admin/multi-user/ (2020), (Accessed on 11/01/2020)

[5] Android: Application Fundamentals — Android Developers. https://developer.android.com/guide/components/fundamentals (May 2021), (Accessed on 05/05/2021)

[6] Android: Building multiuser-aware apps. https://source.android.com/devices/tech/admin/multiuser-apps (May 2021), (Accessed on 05/05/2021)

[7] Android: Data and file storage overview — android developers. https://developer.android.com/training/data-storage (June 2021), (Accessed on 06/25/2021)

[8] Android: Permission — Android Developers. https://developer.android.com/guide/topics/manifest/permission-element (May 2021), (Accessed on 05/05/2021)

[9] Android: Permissions updates in android 11 — android developers. https://developer.android.com/about/versions/11/privacy/permissions (Sep 2021), (Accessed on 09/19/2021)

[10] Android: App-ops. https://android.googlesource.com/platform/frameworks/base/+/refs/heads/android11-d1-b-release/core/java/android/app/AppOps.md#foreground (April 2022), (Accessed on 04/27/2022)

[11] Android: Foreground services — android developers. https://developer.android.com/guide/components/foreground-services (April 2022), (Accessed on 04/14/2022)

[12] Apktool: Apktool—a tool for reverse engineering 3rd party, closed, binary android apps. https://ibotpeaches.github.io/Apktool/ (May 2021), (Accessed on 05/05/2021)

[13] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. Acm Sigplan Notices **49**(6), 259–269 (2014)

[14] Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.R., Shastry, B.: Towards taming privilege-escalation attacks on android. In: NDSS. vol. 17, p. 19. NDSS, San Diego, California, USA (2012)

[15] Davi, L., Dmitrienko, A., Sadeghi, A.R., Winandy, M.: Privilege escalation attacks on Android. In: International Conference on Information Security. pp. 346–360. Springer, Springer, Boca Raton, Florida, USA (2010)
[16] Elsabagh, M., Johnson, R., Stavrou, A., Zuo, C., Zhao, Q., Lin, Z.: {FIRMSCOPE}: Automatic uncovering of {Privilege-Escalation} vulnerabilities in {Pre-Installed} apps in android firmware. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 2379–2396 (2020)
[17] Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Transactions on Computer Systems (TOCS) **32**(2), 1–29 (2014)
[18] F-Droid: F-Droid - Free and Open Source Android App Repository. `https://f-droid.org/`, (Accessed on 05/11/2022)
[19] F-secure Labs: Xiaomi redmi 5 plus second space password bypass. `https://labs.f-secure.com/advisories/xiaomi-second-space/` (May 2021), (Accessed on 05/05/2021)
[20] Felt, A.P., Wang, H.J., Moshchuk, A., Hanna, S., Chin, E.: Permission re-delegation: Attacks and defenses. In: USENIX security symposium. vol. 30, p. 88. USENIX, San Francisco, CA, USA (2011)
[21] Google: IActivityManager source code. `https://android.googlesource.com/platform/frameworks/native/+/refs/heads/android10-c2f2-release/libs/binder/IActivityManager.cpp#82` (May 2021), (Accessed on 16/05/2022)
[22] Hardy, N.: The confused deputy (or why capabilities might have been invented). ACM SIGOPS Operating Systems Review **22**(4), 36–38 (1988)
[23] Heuser, S., Negro, M., Pendyala, P.K., Sadeghi, A.R.: Droidauditor: forensic analysis of application-layer privilege escalation attacks on android (short paper). In: International Conference on Financial Cryptography and Data Security. pp. 260–268. Springer (2016)
[24] Kanonov, U., Wool, A.: Secure containers in android: the samsung knox case study. In: Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices. pp. 3–12. ACM, Vienna, Austria (2016)
[25] Li, R., Diao, W., Li, Z., Du, J., Guo, S.: Android custom permissions demystified: From privilege escalation to design shortcomings. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 70–86. IEEE (2021)
[26] Lu, L., Li, Z., Wu, Z., Lee, W., Jiang, G.: Chex: statically vetting android apps for component hijacking vulnerabilities. In: Proceedings of the 2012 ACM conference on Computer and communications security. pp. 229–240 (2012)
[27] Ratazzi, P., Aafer, Y., Ahlawat, A., Hao, H., Wang, Y., Du, W.: A systematic security evaluation of Android's multi-user framework. arXiv preprint arXiv:1410.7752 **1**(1), 1–10 (2014)
[28] Reardon, J., Feal, Á., Wijesekera, P., On, A.E.B., Vallina-Rodriguez, N., Egelman, S.: 50 ways to leak your data: An exploration of apps' circumvention of the android permissions system. In: 28th USENIX Security Symposium (USENIX Security 19). pp. 603–620. USENIX, Santa Clara, CA, USA (2019)

[29] Samsung Knox: Root of trust. https://docs.samsungknox.com/admin/whitepaper/kpe/hardware-backed-root-of-trust.htm (May 2021), (Accessed on 05/06/2021)
[30] Samsung Knox: Secure folder — samsung knox. https://www.samsungknox.com/en/solutions/personal-apps/secure-folder (May 2021), (Accessed on 05/06/2021)
[31] Stats, S.G.: Mobile & tablet android version market share worldwide — statcounter global stats. https://gs.statcounter.com/android-version-market-share/mobile-tablet/worldwide/#monthly-202006-202009, (Accessed on 08/20/2021)
[32] Sutter, T., Tellenbach, B.: Simple spyware: Androids invisible foreground services and how to (ab)use them. In: Black Hat Europe, London, 2.-5. Dezemeber 2019. p. 27. Black Hat Europe, London, UK (2019)
[33] Tuncay, G.S., Demetriou, S., Ganju, K., Gunter, C.: Resolving the predicament of android custom permissions. Network and Distributed System Security Symposium **1**(1), 1–15 (2018)
[34] Xing, L., Pan, X., Wang, R., Yuan, K., Wang, X.: Upgrading your android, elevating my malware: Privilege escalation through mobile os updating. In: 2014 IEEE Symposium on Security and Privacy. pp. 393–408. IEEE, IEEE, Berkeley, CA, USA (2014)
[35] Xu, Y., Wang, G., Ren, J., Zhang, Y.: An adaptive and configurable protection framework against android privilege escalation threats. Future Generation Computer Systems **92**, 210–224 (2019)
[36] Zhang, M., Yin, H.: AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications. In: NDSS (2014)
[37] Zhong, X., Zeng, F., Cheng, Z., Xie, N., Qin, X., Guo, S.: Privilege escalation detecting in android applications. In: 2017 3rd International Conference on Big Data Computing and Communications (BIGCOM). pp. 39–44. IEEE (2017)
[38] Zhou, Y., Jiang, X.: Detecting passive content leaks and pollution in android applications. In: Proceedings of the 20th Network and Distributed System Security Symposium (NDSS). pp. 1–16. Association for Computing Machinery New York NY United States, Bangalore India (2013)

9 Appendix

Vendors		Google Multi-user	Samsung Folder 1.2	Samsung Folder 1.4	Huawei Xiaomi
Activity Manager	#1 Access broadcast receiver/service	●	●	●	◐
Content Provider	#2 Access Content provider	●	●	●	●
Media Content Provider	#3 List images	●	●	●	●
	#4 Insert images	●	●	●	●
	#5 Delete images	●	●	●	●
	#5 Read images	●	●	○	●
	#6 Write images	●	○	○	●
Package Manager (PM)	#9 List applications	●	●	●	●
	#10 Uninstall applications	●	●	●	●
	#11 Pull applications	●	●	●	●
	#12 Install applications	●	○	○	●
	#13 Grant/Revoke permissions	●	●	●	●

Table 1. MU attacks across vendor implementation. (●) exploited; (○) N/A; (◐) untested