Journal of Hardware and Systems Security (2019) 3:45–63
https://doi.org/10.1007/s41635-018-0053-2

Unravelling Security Issues of Runtime Permissions in Android

Efthimios Alepis[1] · Constantinos Patsakis[1]

Received: 20 May 2018 / Accepted: 15 October 2018 / Published online 25 October 2018

Abstract
Mobile computing is conquering human-computer interaction and user Internet access over the last years. At the same time, smartphone devices are equipped with an increasing number of sensors, realizing context awareness, while accompanying their users in their daily life. As a result, these highly sophisticated and multi-modal devices deal with a surprisingly big amount of data, much of which is private and sensitive. To control data access, OSes have special permission mechanisms, often controlled by the users. The Android permission model has radically changed over the last years, in an effort to become more flexible and protect its users more effectively. This work presents a thorough analysis of the new android permission architecture, accompanied with a criticism regarding its advantages and disadvantages based on a number of disclosed security issues.

Keywords Android · Security · Permissions · Privacy · Smartphones

1 Introduction

In an era where the world's most valuable resource is data [21], replacing oil, the context under which data exploitation is taking place is perhaps one of the most plausibly important discussions. Both companies and humans, individually, have always been trying to maximize their profit. The latter in the digital world has been realized in a clash of data harvesting. Unsurprisingly, data originating from users has been labeled as very valuable [46] and more specifically, regarding smartphones, users are paying less money using mobile apps, since they are becoming the actual product [31]. As a result, in modern life, we are facing a reality where developers, and correspondingly companies, want as much access as possible to user data, while concurrently, users want to protect their privacy and their rights, with various emerging policies around the globe [23]. Mobile operating systems are lying arguably somewhere "in the middle" since they want to satisfy both parties and fulfill their claims, even though their interests seem to be in many cases conflicting.

✉ Constantinos Patsakis
kpatsak@gmail.com

1 Department of Informatics, University of Piraeus, 80, Karaoli, Dimitriou, 18534, Piraeus, Greece

Android is the most widely used platform for hand-held devices having a huge user base in the scale of billions. While the core of Android is Linux, the platform has been radically redefined by Google to meet the specific needs of the users in devices with constrained resources. Android as well as iOS are almost a decade old and entered the market when other operating systems were monopolizing. Nonetheless, they quickly conquered the market, currently owning more than 90% of the market share. This quick shift in the market can be attributed to a big transformation in the functionality that both operating systems allowed: the installation of third-party applications. Companies and independent developers quickly started developing mobile applications for both these platforms exploiting the new capabilities that these devices are equipped with, creating a new ecosystem.

Evidently, over the last years of smartphone development, modern mobile devices needed more fine-grained security models for their users since very sensitive data were handled, other can be extracted from the embedded sensors and most importantly, because these devices are being constantly used in most peoples' everyday life. As a result, users who were inexperienced with computers, as well as children and elderly are nowadays smartphone users. Therefore, it is easier to trick modern users who are also more prone to experience security and privacy threats. To address these issues, a security model that would inform users about permission settings only once during application installation has been considered as insufficient. Unfortunately, the main reason for this was the fact that many users did not take these permission settings into consideration and this has often resulted in them being misled or even deceived. An improved model that would handle security and privacy threats during runtime for stock devices was first introduced in iOS 7 from Apple, and in Android Marshmallow from Google. The intention of these models is to provide the required information and request permissions during applications' runtime once the application needs to access them. Therefore, an application must not only declare which permissions are necessary for its incorporated functions, but users would also have a broader view of how their sensitive data are handled whenever and even each time these data were handled by the application. Hence, access to the respective resources and sensors has the explicit permissions granted by the user.

Having used the new security model of Android, developers have significantly changed their programming logic, since a large amount of smartphone applications depends on the granted permissions to interact with the operating system's environment. As a next step after the incorporation of the new permission "logic," an evaluation of the first results of the new model must follow to determine whether the applications' behavior has been improved compared with the older model. In this sense, we have developed some smartphone applications that would be used as testbeds for the new security model.

The new permission framework introduced in Android Marshmallow is just a few years old and, evidently, continuously changing, yet the model remains the same. This research effort mainly focuses on providing ground evidence about its problems and possible ways to overcome some of them. The presented security flaws have also been tested against the most recent Android OS versions to date, namely Android Nougat and Android Oreo. Additionally, the paper includes related privilege escalation security issues, presenting attacks that require zero-permission apps, while, finally, the apps possess either dangerous or even system-level permissions, making them potentially very "dangerous" for their users.

Main Contributions The goal of this work is to go a step before the implementation, and discuss the actual permission model of Android, which essentially describes how sensitive information is handled by applications. In this regard, the contributions of this work are twofold. First, we provide a detailed description of Android's permission model, in an effort to analyze the needs that led to its introduction. Further to just stating the new features of the security model, we proceed to an in-depth analysis of new problems we have identified and others that are augmented. To validate our claims, we have developed a number of test applications, some of which are used for comparison, while others expose serious security and privacy issues. It should be highlighted that the recent changes in Google Play[1] make the usage of this model mandatory for all apps. This manuscript is an extended version of [2]. In this paper, we focus on more thoroughly investigating the Android permission architecture, detailing how the permissions are applied. Moreover, we disclose some important security issues, reported and acknowledged by Google. These security issues include, but are not limited to, transformation attacks, usage of other application's resources, and privilege escalation. Finally, we update the literature with other attacks that were possible due to the flawed implementation of the permission model, e.g., [44].

Road Map The rest of this work is structured as follows. We first provide a brief overview of Android platform and related work in Section 2. Then, in Section 3, we detail the new permission model in Android and how it is applied. Section 4 highlights several fallacies of the model and the risks that users are exposed to, while it also includes some new attacks that result in serious permission escalation issues. Section 5 discusses remedies for the security issues that are presented in this work. Finally, the article concludes with some remarks and ideas for future work.

2 Related Work

Each application in Android is installed through an APK file which is a compressed package that contains everything that the application needs to be executed, such as the bytecode, resource files, and metadata. This APK is installed by invoking the Package Manager system service. The official and most widely used method is through *Google Play*, an application with elevated permissions which connects to Google's application market and downloads the requested applications enabling also the indexing and a reputation system. Users may also invoke the Package Installer by downloading or copying an application to the phone storage and asking Android to open it. Finally, advanced users may use Android Debug Bridge (ADB) through their computers, a tool provided by Google through Android SDK which offers developers many additional functionalities. If a user selects to install an application using the first two scenarios, he will be presented with a screen which notifies him which permissions have to be granted in order for the app to run in his device. Up to Marshmallow, the user was forced to either accept the proposed permissions and install the application, or to reject them, thus canceling

[1]https://developer.android.com/distribute/best-practices/develop/target-sdk

the entire installation process. Therefore, permissions were given once to application and they could not be revoked, unless the user decided to uninstall the application.

This “take-it-or-leave-it” policy was also present in iOS; nevertheless, modding communities introduced fine-grained policies in rooted devices[2][3]. In this case, users were allowed to explicitly revoke access on installed applications; nonetheless, this quite often affected the stability of these apps. Therefore, while users gained more control of their devices and data, it reduced the quality of user experience.

When Android and iOS first introduced their application markets, developers were not requesting a big number of permissions. However, developers and companies seemed to start realizing that the capabilities of these devices, such as location awareness, access to contact lists, usage patterns, etc. could provide them a great wealth of information that could be exploited and therefore monetized, and in a sense became more “greedy,” as apps constantly request more and more permissions as time passes [15, 42]. In many occasions, simple applications require absurd permissions only to harvest user data [41]. While in some instances, this can lead to malicious acts, unfortunately it has also paralyzed user reflexes towards such requests. As shown by Felt et al. [28], the continuous increase of application permission requests for access to sensitive permissions or unrelated to their core functionality, made users ignore Android warnings and install apps without understanding to what risks they expose themselves, a result which was also verified in other cases [9, 35].

While the Android permission model is considered secure, from time to time, several severe security issues have appeared. For instance, Davi et al. [17] showed that they could escalate the access privileges of applications by performing calls to other applications, which had already granted the respective privileges. Since Android does not perform checks on transitive calls, they managed to create a chain of calls between applications, so that the combination of the applications gained the necessary privileges. In the scenario of Davi et al. a malicious application exploits the vulnerabilities of a legitimate application to execute the malicious code. Since the malicious application was not initially granted the privileges to perform an action, it is most likely to evade the respective checks and continue its nefarious acts. Orthacker et al. [37] extended the aforementioned scenario to show that an adversary could use *permission spreading*, that is split the necessary privileges to different applications, and through intercommunication launch the attack. Similar approaches regarding app collusion and spread of permissions have been reported by Dimitriadis et al. [19] as well as Blasco and Chen [12].

[2]http://repo.xposed.info/module/biz.bokhorst.xprivacy

[3]https://lineageos.org/

On top of the aforementioned issues, Grace et al. [33] found that many applications would use plain HTTP to download code and execute it on client’s device. While insecure, this method is often seen in Android applications [40], allowing an adversary to alter the downloaded file during its transfer. This is amplified by an inherent threat to all Android applications: The access level to a resource is granted *per application* and *not per component*. Theoretically, this does not raise any important issue, since all the components are handled by the same entity, the developer who created the app. However, as discussed in the following paragraphs, this is not always the case.

In general, recent statistical and research findings indicate that ads have started becoming more and more “greedy” regarding users’ information. In fact, the more installations an app has, the more privacy invasive it tends to be [14]. Apart from direct requests to get users’ location, ad libraries, may perform additional WiFi scans to determine users’ location, scan whether a user has accounts in social networks, or even scan the device to find which applications have been installed [13]. Recently, more advanced ad libraries manage to link devices by playing inaudible sounds from one device and collecting them from the microphone of mobile devices that use applications where such an ad library has been embedded [30].

Yang et al. [47] attempted to crowdsource users’ permissions preferences in a semi-automatic way. Their application, Droidganger, executed an application monitoring its behavior, and gradually revoked permissions to determine problems with its execution. When issues appeared, the user would be prompted to comment them, and the results were aggregated in a central server. This approach could potentially remove unnecessary permissions from applications which requested far more permissions than they actually needed, which accounts for a huge market share and whose acts could be characterized as malicious [13, 45]. To decrease user interaction, automated approaches have been introduced such as the work of Bartel et al. [11] and Tsiakos’ and Patsakis’ [43], with the latter aiming towards advertisement networks.

To provide more fine-grained permissions, Jeon et al. [34] developed some tools which can be used to detail which permissions are granted to an application and which are not, in order to make them comply with the principle of least “permissions.” For more on Android and the old permission model, the interested reader may refer to [25, 26].

In view of the above, Android Marshmallow introduced many changes in Android, many of which are focused on the security and privacy of the platform. The new version has drastically decreased the number of dangerous permissions, which had reached more than 260 in API level 22. Some of these permissions are illustrated in Table 1. Contrary to the line of previous versions where, as highlighted by Wei

Table 1 Some of the Android permissions in API level 22

Permissions		
ACCESS_ALL_EXTERNAL_STORAGE	ACCESS_LOCATION_EXTRA_COMMANDS	ACCOUNT_MANAGER
ACCESS_CACHE_FILESYSTEM	ACCESS_MOCK_LOCATION	ALLOW_ANY_CODEC_FOR_PLAYBACK
ACCESS_CHECKIN_PROPERTIES	ACCESS_MTP	ASEC_ACCESS
ACCESS_COARSE_LOCATION	ACCESS_NETWORK_CONDITIONS	ASEC_CREATE
ACCESS_CONTENT_PROVIDERS_EXTERNALLY	ACCESS_NETWORK_STATE	ASEC_DESTROY
ACCESS_DRM_CERTIFICATES	ACCESS_NOTIFICATIONS	ASEC_MOUNT_UNMOUNT
ACCESS_FINE_LOCATION	ACCESS_PDB_STATE	ASEC_RENAME
ACCESS_FM_RADIO	ACCESS_SURFACE_FLINGER	AUTHENTICATE_ACCOUNTS
ACCESS_INPUT_FLINGER	ACCESS_WIFI_STATE	BACKUP
ACCESS_KEYGUARD_SECURE_STORAGE	ACCESS_WIMAX_STATE	BATTERY_STATS

et al. [45], the permissions were made to satisfy vendors, the new permission model is focused towards developers. Nonetheless, as it is going to be discussed in the next sections, by no means has the new model become more fine-grained, and more importantly, it cannot be considered transparent. Nonetheless, starting from Marshmallow, the user can revoke permissions or grant them upon request to further refine access rights.

3 The New Permission Model

From the very beginning of Android, in 2007, until Android L (all API levels until 23), application permissions were accepted by users in the first steps of their installation. With this move, Google had moved towards achieving two goals: firstly to inform the user about which operations an application may perform and secondly to mitigate possible attacks by limiting the application access. The permission model provided a "take-it-or-leave-it" approach as users would either accept the requested permissions and install the application, or the application would not be installed. After many years of using this approach, the Android team brought a new model in October 5, 2015, with the introduction of the new Marshmallow flavor. As Google states, Android 6.0 (API level 23) includes a variety of system changes and API behavior changes, improving resource allocation, stability and performance. Nevertheless, probably the most notable change, is the complete redesign of its permission model, which is listed on the top of Google's list of changes as "Runtime Permissions." According to Google's developer site[4], Android 6.0 introduces a new permission model, where users can directly manage app permissions at runtime. It is also highlighted that this new security model gives users improved visibility and control over application permissions.

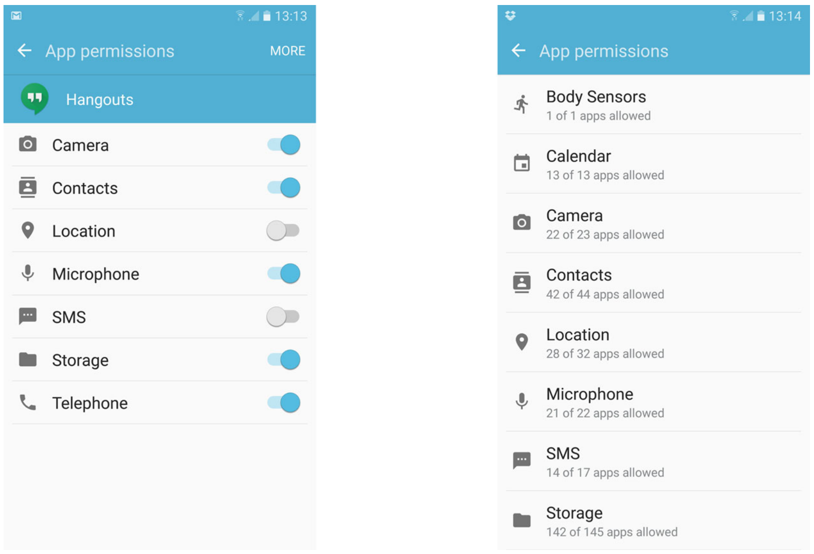
The new permission model allows users to selectively revoke dangerous permissions, Fig. 1a, and facilitates users' privacy control by grouping applications according to granted permissions, Fig. 1b. Moreover, the applications are not granted dangerous permissions during installation, but on runtime, even if they are included in the system image. The new model mandates all apps to check for and request permissions at runtime.

Evidently, as it will also been shown in the next sections, even the biggest security change, introduced in Marshmallow and kept through Android Nougat and Android Oreo, namely "Runtime Permissions" contains serious security flaws, which can be exploited and result in exposing users' security and trust.

Permissions in Android are characterized according to the risk implied when granting them into the following four categories:

– Normal: These permissions can be regarded as the ones that expose the user or the system to the least possible risk when granted. Therefore, the system automatically grants them at installation, without asking for the user's explicit approval.
– Dangerous: In this category, the risk is higher as granting these permissions can expose private user data or allow control of the device. Since these permissions imply a potential high risk, explicit user approval is required to be granted. Typical, such permissions, include access to the microphone, contacts, camera, etc.
– To allow interoperability, Android application may exchange information through inter component communication (ICC). Nonetheless, to guarantee that specific applications are granted some permissions, Google has introduced the *signature* permission. Therefore, Android grants access to an application only if the requesting application is signed with the same certificate as the

[4]https://developer.android.com/about/versions/marshmallow/android-6.0-changes



13:13

App permissions MORE

Hangouts

Camera	● on
Contacts	● on
Location	○ off
Microphone	● on
SMS	○ off
Storage	● on
Telephone	● on

13:14

App permissions

Body Sensors	1 of 1 apps allowed
Calendar	13 of 13 apps allowed
Camera	22 of 23 apps allowed
Contacts	42 of 44 apps allowed
Location	28 of 32 apps allowed
Microphone	21 of 22 apps allowed
SMS	14 of 17 apps allowed
Storage	142 of 145 apps allowed

(a) Changing permissions after installation. (b) An overview of the granted permissions.

Fig. 1 Managing permissions in Marshmallow

application that declared the permission without user notification.

– In order to cater for the needs of applications that are supplied by the manufacturers, Google has introduced the *signatureOrSystem* permission. This permission is granted only to apps that reside in the Android system image or that are signed with the same certificate as the application that declared the permission. Such privileges allow apps to reboot a device or to allow an application to clear the caches of all installed applications on the device. Essentially, this permission is designed for manufacturers.

Apart from the above four main categories, several additional flags can be used to further characterize the protection level of a permission such as *privileged*, also known as *system*, used when multiple vendors have applications built in to a system image to determine who can use what, *installer*, *verifier* for applications which install and verify packages respectively, etc. For a detailed list of these permissions and where they apply, the interested reader may refer to [8]. Moreover, since many unprotected APIs were found in previous versions [36], additional protection mechanisms have been integrated for many intents. Finally, developers may create their own permission groups to expose their functionality to other apps by defining permissions which those other apps can request.

According to Google[5], the categorization of permissions to normal and dangerous implies the existence of a *direct* privacy risk. It is worth noticing that prior to Marshmallow, Android had numerous permissions, part of them shown in Table 1, flooding the user with information. This fact was often exploited by developers who for instance would request many permissions which would not actually be used, so that the user would not be able to see the dangerous ones on top of the landing screen. The normal permissions since API 23 are illustrated in Table 2.

To counter such issues, Android further grouped "dangerous" permissions according to their access level in terms of functionality, as it is illustrated in Fig. 2. These groups facilitate users, as they group permissions according to a specific functionality, e.g., "Manage SMS," instead of granting permissions to each functionality independently, e.g., receive, read, send SMS, the user grants permissions per application to a group of permissions, enabling full access to the rest of the permissions in the same group. Certainly, this approach significantly improves user interaction and experience as users have to respond to significantly less notifications.

Prominently, Google introduced several features in the permission model which are not apparent from the aforementioned description, in order to further protect users' privacy. For instance, since Marshmallow, developers are expected to request ACCESS_FINE_LOCATION or ACCESS_COARSE_LOCATION permissions to access hardware identifiers of nearby external devices via Bluetooth and WiFi scans. This change was introduced to prevent location disclosure, as many applications were trying to exploit this knowledge to correlate this information with the location of already known devices. However, hardware identifiers can still be extracted to locate users, e.g., as shown in [4] unique hardware identifiers can be extracted by the use of WiFi-P2P.

[5] https://developer.Android.com/training/permissions/requesting.html

Table 2 Normal permissions in Marshmallow

Permissions	
ACCESS_LOCATION_EXTRA_COMMANDS	NFC
ACCESS_NETWORK_STATE	READ_SYNC_SETTINGS
ACCESS_NOTIFICATION_POLICY	READ_SYNC_STATS
ACCESS_WIFI_STATE	RECEIVE_BOOT_COMPLETED
BLUETOOTH	REORDER_TASKS
BLUETOOTH_ADMIN	REQUEST_IGNORE_BATTERY_OPTIMIZATIONS
BROADCAST_STICKY	REQUEST_INSTALL_PACKAGES
CHANGE_NETWORK_STATE	SET_ALARM
CHANGE_WIFI_MULTICAST_STATE	SET_TIME_ZONE
CHANGE_WIFI_STATE	SET_WALLPAPER
DISABLE_KEYGUARD	SET_WALLPAPER_HINTS
EXPAND_STATUS_BAR	TRANSMIT_IR
GET_PACKAGE_SIZE	UNINSTALL_SHORTCUT
INSTALL_SHORTCUT	USE_FINGERPRINT
INTERNET	VIBRATE
KILL_BACKGROUND_PROCESSES	WAKE_LOCK
MODIFY_AUDIO_SETTINGS	WRITE_SYNC_SETTINGS

Moreover, to protect users from phishing and ransomware attacks, since Marshmallow, an app has to explicitly request the permission to overlay itself over others. In fact, to indicate the criticality of granting such a permission, Android breaks the usual user interface redirecting the user to another settings menu to grant this specific permission, with indicative screens. Nonetheless, this protection mechanism is rather flawed as it will be discussed later on.

The enforcement of the permission model is a multi-step process, but before we describe this process, we should highlight that each application in Android is considered as a different user; hence, it is granted a different UID. The reason for this choice is to prevent applications from accessing the data and private resources of other installed apps. Once an app performs a call to the framework API, this is accompanied by the UID of the app. The framework then checks whether it has been assigned upon installation in the `AndroidManifest.xml` file. Should this be the case, Android checks the permission level of this call (normal, dangerous etc.). If it is a normal permission it is granted and access to the API is provided. However, if it is a dangerous permission, the system will query whether access to this resource has been granted by the user and allow or deny the access accordingly. Finally, if the permission is

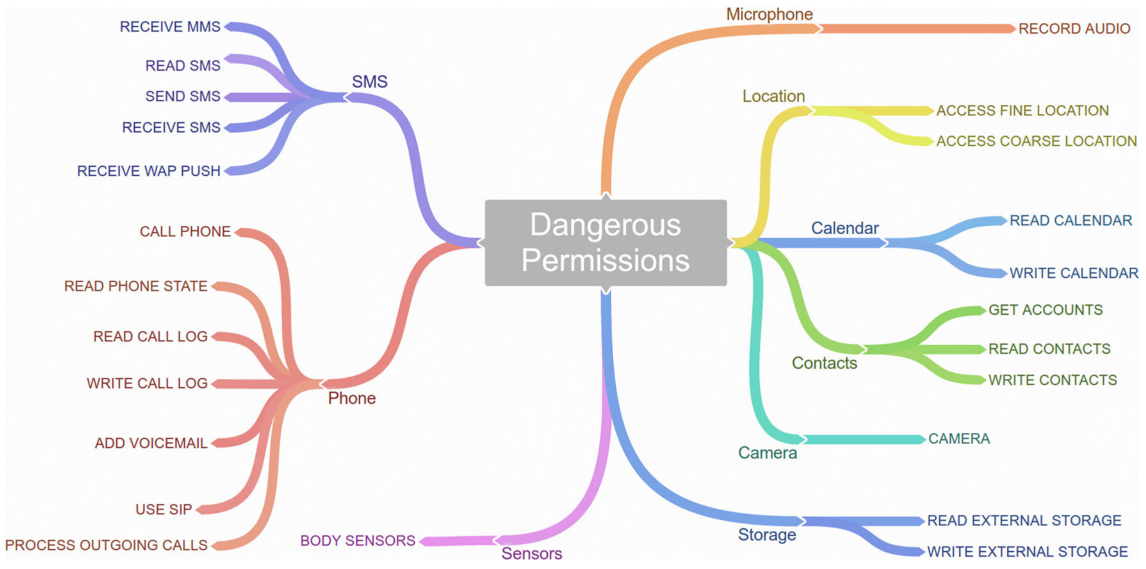


Fig. 2 Dangerous permissions groups

signature or signatureOrSystem, then the system will have to check the signature of the app with the requesting UID before granting the corresponding access.

To further protect the Android ecosystem, Google requires the user to explicitly grant some app permissions through completely different permission management screens (see Fig. 3b, c, d), which differ greatly from the common permission screen supplied for handling dangerous permissions (seen in Fig. 3a). In the former category, we have permissions such as `SYSTEM_ALERT_WINDOW`, `BIND_ACCESSIBILITY_SERVICE`, `WRITE_SETTINGS`, and `PACKAGE_USAGE_STATS`. To prevent users from carelessly granting these permissions, Android provides a completely different interface and in principle the corresponding settings are well-hidden in the menus so that users will grant access only when deemed necessary (see Fig. 3 for comparison). To understand the extent of the risk that these permissions expose their users to, one has to consider that the `SYSTEM_ALERT_WINDOW` allows an application to overlay every Android activity, and therefore can totally deceive the user. The `BIND_ACCESSIBILITY_SERVICE` permission allows an application to imitate user tapping on the screen. Therefore, once granted to an app, it can perform any action on the user's device. Finally, the `PACKAGE_USAGE_STATS` will be discussed in detail in the following paragraphs.

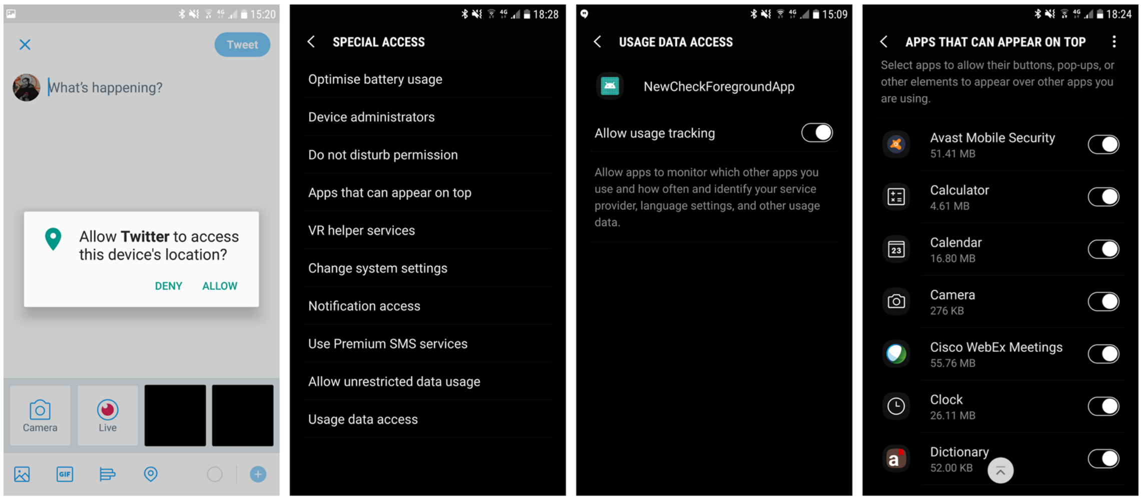
Beyond the aforementioned categorization of permissions, Android has the same Linux-based mechanism for UID/GID-based access control to enforce the permission mechanism. All users and groups are assigned with an ID (see Listing 1). As implied by this code excerpt, apps are assigned an ID above 10000, referred to as AID. Once a user grants a permission to an app, the app is added to the corresponding group, and it can access the respective resource. Therefore, if an app belongs to groups 1006 and 1021, it can access camera and GPS.

4 Drawbacks of the New Permission Model

The aforementioned permission categorization may seem quite secure, improving the previous model, as sensitive information seems to be protected and selectively disclosed. Nonetheless, the implementation of the new model also introduces several drawbacks which are going to be discussed in the following paragraphs.

4.1 Flawed Implementation of the Permission Model

One would expect that after the introduction of Marshmallow, all of the functionality would be immediately provided to the software systems that have the novel permission model installed. However, this is subject to the targeting API of the installed app. Practically, this means that apps may exploit this feature to extend their permissions. For instance, if an app is targeting an "old" API, prior to Marshmallow, then the app will request the permissions on installation, and if the user accepts them, then once the app is loaded, the permissions have been granted, as in the pre-Marshmallow era, so the user has to disable the dangerous permissions manually and no granting screen will be displayed to the user. This, however, seems to contradict the user's initial


15:20
Tweet
What's happening?
Allow Twitter to access this device's location?
DENY
ALLOW
Camera
Live
18:28
SPECIAL ACCESS
Optimise battery usage
Device administrators
Do not disturb permission
Apps that can appear on top
VR helper services
Change system settings
Notification access
Use Premium SMS services
Allow unrestricted data usage
Usage data access
15:09
USAGE DATA ACCESS
NewCheckForegroundApp
Allow usage tracking
Allow apps to monitor which other apps you use and how often and identify your service provider, language settings, and other usage data.
18:24
APPS THAT CAN APPEAR ON TOP
Select apps to allow their buttons, pop-ups, or other elements to appear over other apps you are using.
Avast Mobile Security 51.41 MB
Calculator 4.61 MB
Calendar 16.80 MB
Camera 276 KB
Cisco WebEx Meetings 55.76 MB
Clock 26.11 MB
Dictionary 52.00 KB

(a) Granting a dangerous permission. (b) Managing some system permissions. (c) Allowing an app to track usage of other installed apps. (d) Managing the permission to overlay other apps.

Fig. 3 Different interfaces for managing permissions in Android

Listing 1 Excerpt from available UIDs/GIDs in Android as defined in AOSP source code [32]

```
#define AID_ROOT 0 /*traditional unix root user*/
/*The following are for LTP and should only be used for testing*/
#define AID_DAEMON 1 /*traditional unix daemon owner*/
#define AID_BIN 2  /*traditional unix binaries owner*/
#define AID_SYSTEM 1000 /*system server*/
#define AID_RADIO 1001 /*telephony subsystem, RIL*/
#define AID_BLUETOOTH 1002 /*bluetooth subsystem*/
#define AID_GRAPHICS 1003 /*graphics devices*/
#define AID_INPUT 1004 /*input devices*/
#define AID_AUDIO 1005 /*audio devices*/
#define AID_CAMERA 1006 /*camera devices*/
#define AID_LOG 1007 /*log devices*/
#define AID_COMPASS 1008 /*compass device*/
#define AID_MOUNT 1009 /*mountd socket*/
#define AID_WIFI 1010 /*wifi subsystem*/
#define AID_ADB 1011 /*android debug bridge (adbd)*/
#define AID_INSTALL 1012 /*group for installing packages*/
#define AID_MEDIA 1013 /*mediaserver process*/
#define AID_DHCP 1014 /*dhcp client*/
#define AID_SDCARD_RW 1015 /*external storage write access*/
#define AID_VPN 1016 /*vpn system*/
#define AID_KEYSTORE 1017 /*keystore subsystem*/
#define AID_USB 1018 /*USB devices*/
#define AID_DRM 1019 /*DRM server*/
#define AID_MDNSR 1020 /*MulticastDNSResponder (service discovery)*/
#define AID_GPS 1021 /*GPS daemon*/
#define AID_UNUSED1 1022 /*deprecated, DO NOT USE*/
...
#define AID_APP 10000 /*TODO: switch users over to AID_APP_START*/
#define AID_APP_START 10000 /*first app user*/
#define AID_APP_END 19999 /*last app user*/
```

intuition. Despite the obvious issues that this approach raises with normal and dangerous permissions, the flawed implementation of permission handling introduces further security issues, far more dangerous.

One of the most well-known examples is the "notorious" SYSTEM_ALERT_WINDOW permissions. This permission, introduced in Marshmallow, is a system permission, and theoretically can be granted only through specific user interaction. This permission can be considered as "really dangerous" as it allows an adversary to create overlays that can deceive the user by covering any part of the screen while allowing the rest of the screen (the overlayed app) to be responsive. Fratantonio et al. [29] used this trick to lure users to grant almost full control of the device to them by afterwards tricking the user to grant them the BIND_ACCESSIBILITY_SERVICE permission. The latter permission, also system permission, allows an app to imitate user interaction, e.g., user clicks, in every Android activity. However, the overlay issues have been proven to be far more severe and may not even require system permissions [5].

Recently, another severe vulnerability was disclosed by Tuncay et al. [44]. The researchers actually use a transformation approach to trick the system to grant them dangerous permissions. To do that they first create an app which requests a custom permission with the protection level normal or signature and set this to be a part of a system permission group. Once the user installs this app, these permissions are granted to the app. However, they are incorrectly flagged by the system; therefore, an adversary may later update the app to request dangerous permissions within the scope of the custom permission, and they will be automatically granted without any kind of user interaction.

4.2 Transformation Attacks in Runtime Permissions

In what follows, we illustrate another attack vector which is based in exploiting apps' actual names. Nevertheless, in this case, it is not used to steal users' private data, by luring them to use a fake application, as reported by the authors in [38], but to enable a malicious app to gain access to all available dangerous permissions, deeply deceiving Android users by exploiting the newly introduced runtime permission model. The problem is amplified by the alarming fact that the malicious app does not only manage to successfully deceive its users, but furthermore, it breaks the users' trust to their already installed, legitimate apps. As a result, in the next paragraphs, it will be shown that a malicious app successfully escalates privileges and also indirectly "puts the blame" on other official market apps for its illegal acts. This security issue has been responsibly disclosed to Google and has been subsequently acknowledged (Android ID 72708800), while its fix has not been released yet, at the time of writing, after more than 7 months of the initial report. Since the last Android version, code named "Pie" has been released in August 2018 and a patch is not yet released, all Android versions, including Marshmallow, Nougat, and Oreo are affected by the disclosed security issue. The attack is presented in the following use case scenario:

– A zero-permission app, named "Bob" is uploaded to Google Play.

- Users download, install and run the app at least once. It requests no permissions, acts just any benign app.
- The attacker issues an update for “Bob.” In this update, the attacker renames the app and requests a number of dangerous permissions. For the sake of simplicity, we assume that the new name is “Google Maps.”
- The update is pushed probably during night (accompanied presumably with other app updates too), when the device is left unattended, e.g., to be charged.
- After the update, the malicious app is able to ask the user to grant it several dangerous runtime permissions to the “former” Bob app. This action is only initiated if the malicious app has found the target app (Google Maps in our use case) installed, through obtaining a list of all the available packages (information which is available to every installed app).
- After a successful attack the previous steps can be repeated, to target other famous apps, and request permissions on their behalf as well.
- Having successfully succeeded in the attacks, the malicious app may change its name back to “Bob,” in order to fully “cover” its malicious actions, escalating privileges and gaining access to dangerous permissions on demand.

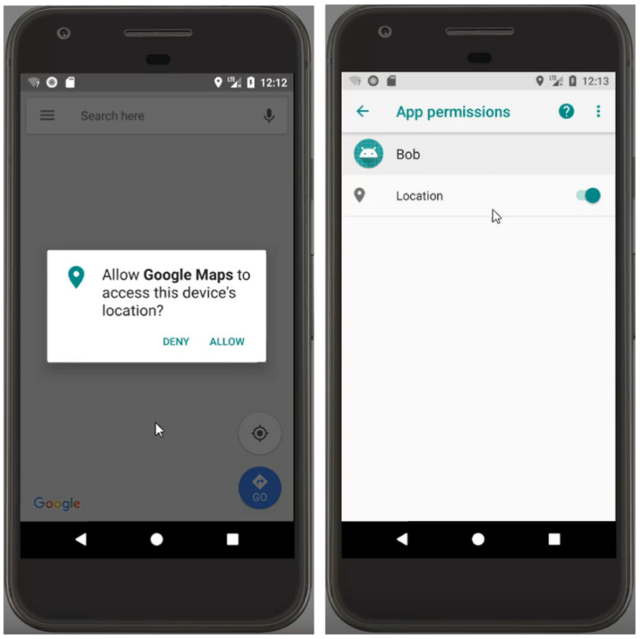
It is important to mention that this attack also involves a zero-size activity (totally invisible to the user) that revokes the runtime system permission. This activity is initiated to show the “fake” runtime permission system dialog to the user and is never actually used. A decent explanation could easily accompany all these permission requests. Moreover, users are clearly aware of the reasons why the legitimate apps would need the dangerous permissions and trust them so they would not have second thoughts to grant them the permissions once again, or even give access to other dangerous permissions.

Our attack scenario supplementarily includes possible time-frames for the attacks to be launched, in order to illustrate the extend and the context in which such use cases could take place:

- Broadcast events, such as device reboot, charging, network changes, etc.
- Forged notifications from (acting as the actual legitimate) apps, as presented by the authors in [38].
- Device unlock: Whenever a runtime permission is requested, even if a device is locked, the runtime dialog will appear to the user as soon as s/he unlocks the device.
- When other, targeted apps have been upgraded, e.g., through periodical version code (`packageInfo.versionCode`) checking.
- Not considering our zero-permission approach, alternatively one can choose to utilize the `UsageStatsManager` service to request the dangerous permissions when a target app is opened by the user, by determining the foreground app. Practically, this means that an app which is granted the `PACKAGE_USAGE_STATS` permission can escalate privileges for all dangerous permissions.

A representative use case is also illustrated in Fig. 4, where a malicious app is requesting a dangerous permission, namely location, forging the genuine app Google Maps. In this hypothetical scenario, the user considers granting access to Google Maps; nevertheless, the dangerous permission is granted to the malicious app.

It should also be noted that in our attack scenario, there is no need to partially overlay an app or a settings dialog as in [5] and [29], but we actually underlay the app. The overlay with the permissions dialog that finally causes the permission escalation security flaw actually belongs to the System UI itself. Another important note is that the names of apps in Google Play are not adequately monitored, allowing them to change names whenever the developer wants. As a result, similar names can be used to trick the users. Moreover, users do not have auditable trails of the updates that apps make. Therefore, users cannot track which changes are made to their apps and when. Unfortunately, even the name and visual identity of the apps may change without any form of notification to the user. In



Allow **Google Maps** to access this device's location?

(a) Malicious app requesting runtime permission (b) Malicious app possessing dangerous permission

Fig. 4 Malicious app runtime permission transformation attack

our independent research, we have confirmed the existence of apps in Google Play that share the same name, e.g., “Dictionary,” while at the same time we have successfully uploaded our own apps in Google Play after having renamed them with other popular apps’ names.

We argue that the illustrated attack exploits the users’ inherent trust in Android mechanisms with the most profound one, users’ trust on Google Play. The presented security flaw that results in permission escalation is amplified by the fact that other genuine apps with perhaps millions or even billions of installations have indirectly been involved in a way that could affect their “presence” in the entire market, potentially resulting in negative side effects such as app unistallations. In fact, if Google did not allow apps to have the same name as other apps, or use the resources of other installed apps, as it is illustrated in a following subsection of the paper, this attack vector would not exist.

4.3 Lack of Control in Running Apps and On-Boot Apps

This part focuses on permission issues related to running applications. More specifically, we discuss about security issues that are related with the apps actively running on an Android device and secondly about issues that arise from apps that are initiated after device boot.

Using the term “running” apps in Android clearly needs a more thorough explanation. Our intention in using the term is twofold, on the one hand, providing an indication whether third party apps are actually “running” and thus having the ability to create a list of running apps and processes. On the other hand, indicating the users’ “weakness” in essentially controlling which apps are running in their devices, when and under which circumstances.

Analyzing the first, the ability to determine the exact identity of running processes within the mobile OS, has caused several threats to arise in the past [5, 16]. These security issues stem from the fact that being able to determine the running processes and more precisely finding the foreground app; the one the user interacts with, paves the way for a variety of malware to launch phishing attacks. Moreover, this allows an adversary to collect valuable usage statistics without the user’s knowledge nor consent.

The second question regarding “running” apps in the Android ecosystem involves user control in third-party apps. Perhaps, the majority of Android permissions focuses on providing or denying access to specific resources within the mobile device. A smaller set of permissions deals with user interaction, such as notification permissions, drawing over other apps and device “wake-locks.” Nevertheless, another significant vector lying more in the dimension of “user control” over the mobile device deals with the users’ ability to actually control the apps’ behavior in terms of when and in which “situations” the installed apps could or should operate. Notably, there are cases where an attempt towards this direction has been made from the OS, as with the `RECEIVE_BOOT_COMPLETED` permission, where apps declare their intent to watch for device restarts, to proceed to further actions. Nevertheless, this permission is a “Normal” one and as a result it remains invisible to the user in the post Marshmallow era, and thus cannot be controlled. Other events of user interaction are included in a large list of “Actions,” such as broadcasted intents actions indicating for example whether WiFi has been enabled, or disabled, or the intent filter `android.intent.action.ACTION_POWER_CONNECTED`, for determining the device’s battery current charge status. Such actions, namely a significant number of broadcast events, require no permission at all from the app to declare to be notified. Unsurprisingly, the many Android users have probably faced the situation while not interacting with her mobile device, the “Location” indicating icon starts blinking, beyond doubt indicating that a background process is utilizing location services, logging the user’s current location. Regrettably, in these situations, there are actually no means to determine which app is using that data, since this operation can be easily accomplished through a background service. In this use case, it should be highlighted that the discussed problem does not arise from the actual “use” of the location resource, since this is clearly a declared runtime permission, but from the fact that an almost “arbitrary” number of background tasks can be initiated by apps, which cannot be controlled nor monitored by the user, leaving the “force-stop all apps” action an unrealistic option.

Regarding on-boot apps, Android users have no control on which applications are able to start or initiate a background process during their device boot. This is a quite common feature, found in a number of computer OSes, including Windows, where users can choose which applications will be launched once the OS starts. This is allowed for various reasons, including system slow-down, resource and power management and security. On the contrary, in Android ecosystem there is no such control. Therefore, a malicious app, could register to always receive the boot event and this along with a system DOS attack, resulting in system restart, would lead to an endless restart loop that the user cannot stop. This kind of “lack control” both in running and in “on-boot” apps affects all versions of Android, ranging from Marshmallow to Oreo.

4.4 Using Other apps’ Resources

Using other apps’ resources arbitrarily and without the involved app developers’ consent is another significant security issue, closely related with the Android permission mechanism since at the time of writing it does not provide any mechanism to prevent it. As illustrated in our previous work [38], other apps' resources can be used to create forged notifications. Similarly, forged UIs can result in the transformation of malicious Android activities, replicating genuine ones. Correspondingly, for a number of reasons, including malicious acts, installed apps are able to use the resources of other installed apps, with no protection mechanisms present to date. Legitimate apps resources include logos and copyright material, and even apps' compiled source code that should by no means remain "public" and unprotected to third parties for both legal and ethical reasons.

Taking this specific permission issue a step further, we additionally report a related issue regarding the purchased apps from Google Play. In this particular case, we focus on the purchased apps for two basic reasons:

- From the users' perspective, having purchased something is something they own and that they might not want to share with others, especially without their authorization.
- From the developers' perspective, having their App in an App Market as a paid product and not for free, can be plausibly considered as a protection and restriction mechanism towards arbitrary use.

Alarmingly, every installed app has unrestricted access not only to the paid apps resource files, as discussed earlier, but also to the app's installation files (APKs) without requesting any permission, not even the most obvious one: `Storage`. It is worthy to note that even if someone had access to the device's storage, this would involve the SD card and not the `/data` folder where the APKs are stored. Indeed, we have clearly proven and consequently responsibly disclosed to Google that "all apps have access to paid app installation files requiring no permission at all." Secondly, we have further shown that an app can start by declaring no permissions and move on by requesting normal permission such as "Internet," without users knowing about that. As a result, having actual access to a private, paid apk and a "open channel" to unrestrictedly communicate it may lead to "unfortunate" situations. Firstly, "anyone" is able to distribute through the internet and consequently install paid apks. Secondly, installed apps and their developers are also able to view and examine the paid apk's private source code and other private resource files on their own free will. Paid app and source code uncontrollable distribution are some of the most profound consequences of this issue.

In fact, allowing third parties to collect these APKs, which can then be used to be installed in other devices, most likely violates the developers' license agreement and cannot be monitored. However, Google's Android Team considers that "*at the moment installed APKs are not considered private*." Nevertheless, they also stated that "they may consider changing this policy in an upcoming major release of Android," after our report. Correspondingly, since there is no change in this direction even by the release of Android Pie, all Android versions, ranging from Marshmallow to Pie are affected by the disclosed security issue.

4.5 Privilege Escalation via Intents

As already discussed, Kywe et al. [36] identified plenty of unsecured APIs that could be taken advantage of, allowing applications without permissions to exploit them. Since then, the new versions of Android have introduced security mechanisms to address these issues. However, the authors have identified and responsibly disclosed to Google that there are even more security flaws which may arise from the malicious and unrestricted usage of Android intents. For instance, to access the microphone, the dangerous permission `RECORD_AUDIO` needs to be granted; nonetheless, an adversary can use an intent to launch the Speech-to-Text API and automatically convert all microphone input to text without requesting any permission. The latter can be used in combination with the Text-to-Speech API or simple audio to execute arbitrary commands on the device via intents to Voice Assistants [3], extending and automating the attacks of [18]. Evidently, such a privilege escalation attack to access users' voice by the microphone can be exploited for public surveillance. In response to these issues, in the upcoming version of Android Google plans to prevent apps from using the microphone or camera when they are in the background.[6]

4.6 Transparency and Lack of Control

Inarguably, both user interaction and user experience are improved due to the introduction of the runtime permission model. Nonetheless, we argue that the current implementation lacks in terms of transparency and fails to provide fine-grained control to its full potential. Additionally, we argue that the process of granting permissions during runtime does not necessarily improve the previous state where permissions were granted prior to app installation. The following subsections illustrate how the new permission model may negatively affect the user interaction and also specific use cases where the new model lacks in transparency and lack of control regarding its permissions.

[6]https://www.theverge.com/2018/3/7/17091104/android-p-prevents-apps-using-mic-camera-idle-background

4.6.1 Delusive user Interaction

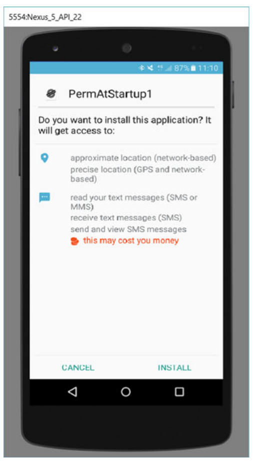
User experience (UX) and flawless user interaction are considered as top priority aims in the entire mobile development process. Through our independent research, we have found realistic use cases where the introduced changes in the OSes permission model have resulted in misleading the users. To validate our claims, we experimented by developing PoC applications which enable us to compare the pre-Marshmallow model to the post Marshmallow one. First, we start the comparison with the previous permission model trying to indicate the actual changes during the installation and execution of some applications. To this end, we developed an application which requests a small number of security permissions, more precisely, permissions to receive, read and send SMSs, and also the permissions to access the location of the device through fine and coarse location permissions. For the evaluation, we created two versions of the application, one that is targeted to API level 22 (Android 5.1) and one that is targeted to API level 23 (Android 6). Consequently, the API 23 app has also been tested against Android Nougat and Android Oreo, giving the same results.

In API level 22, during the installation process, the user is prompted to accept all the permissions in order to proceed with the installation, Fig. 5a. Having successfully acquired the device's location, the application runs smoothly, Fig. 5b, and the user can later review the permissions that are granted to the application, Fig. 5c, without of course being able to revoke them.

Similarly, we perform the same actions using API level 23, which incorporates the new permission model. Notably, Fig. 5a shows that there is no permission required to proceed with the application's installation. Even more interestingly, a user facing this screen is informed that "This application does not require any special access." This information is already misleading and faulty since the application really requires some "dangerous" permissions; however, they are going to be requested after the installation, to be discussed afterwards. Figure 6b, c illustrates the first launch of the application where two groups of permissions are deliberately requested. Perhaps, the most important thing to notice in this process is the fact that these two permissions have not been requested by the application when they were really needed during runtime, but surprisingly during an unrelated to them event, namely the application's first launch. Apparently, granting permissions is not performed on usage request, but when the application is executed. As illustrated in screenshot of Fig. 5b, the mobile application uses both location and SMS features in the corresponding events of two buttons. The application successfully acquires the devices exact location; nevertheless, no pop-up window or alert message informs the user that location service is being used by the application. Finally, Fig. 6d illustrates the application's settings, in the application manager. The aforementioned experiment, after being launched in Android API level 23, has also been successfully evaluated for Android versions 24, 25, 26, and 27.

Having carefully examined the above two use cases where the same application is running targeted to API level 22 and 23 respectively, some important issues arise. Firstly, newer applications, targeted to APIs equal to or greater than 23 do not inform users properly about permissions during installation. Clearly, the information that users receive during installation that they do not require any special access can be considered as either unnecessary or misleading. Moreover, the timing of appearance of the alert requesting a specific permission is also misleading. One would expect that a permission notification would appear once an application tries to access a resource related to that permission. However, applications may ask the user to grant them access to dangerous permission groups on unrelated

Fig. 5 API level 22


5554:Nexus_5_API_22

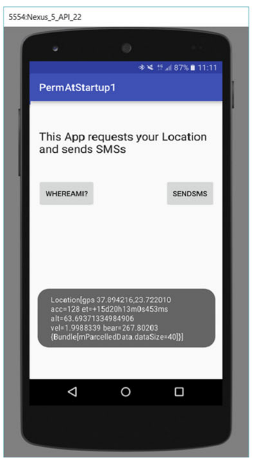
PermAtStartup1

Do you want to install this application? It will get access to:

approximate location (network-based)
precise location (GPS and network-based)

read your text messages (SMS or MMS)
receive text messages (SMS)
send and view SMS messages
this may cost you money

(a) Application installation.

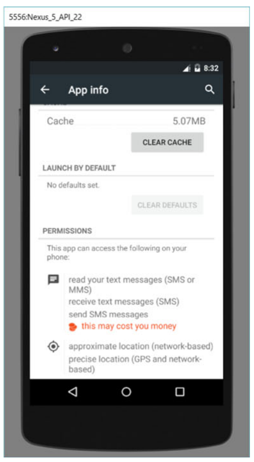

5554:Nexus_5_API_22

PermAtStartup1

This App requests your Location and sends SMSs

```
Location[gps 37.894216,23.722010
acc=128 et=+15d20h13m0s453ms
alt=63.69371334984906
vel=1.9988339 bear=267.80203
{Bundle[mParcelledData.dataSize=40]}]
```

(b) Application running.


5556:Nexus_5_API_22

App info

Cache 5.07MB

LAUNCH BY DEFAULT

No defaults set.

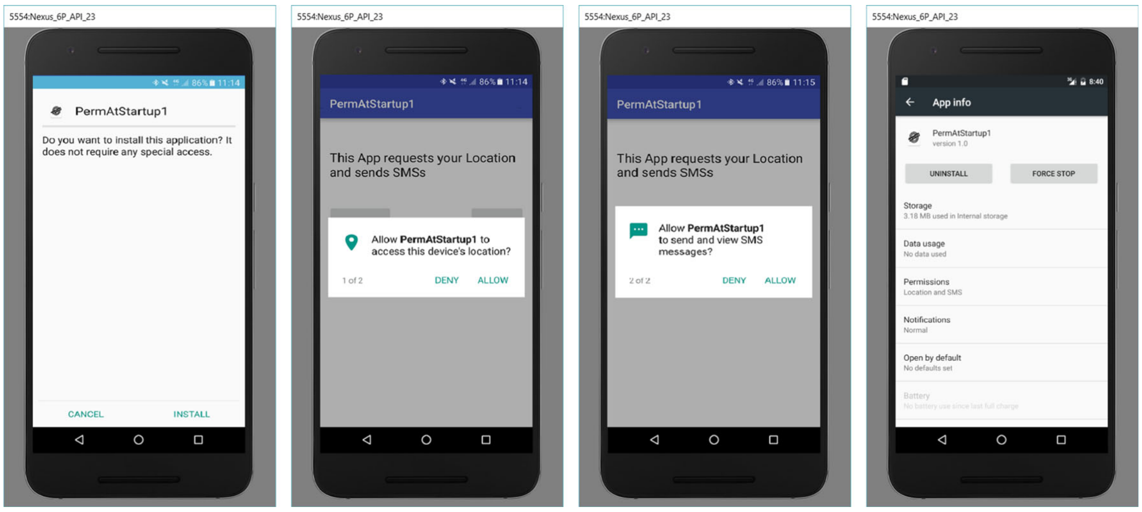
PERMISSIONS

This app can access the following on your phone:

read your text messages (SMS or MMS)
receive text messages (SMS)
send SMS messages
this may cost you money

approximate location (network-based)
precise location (GPS and network-based)

(c) Application info.



Do you want to install this application? It does not require any special access.

This App requests your Location and sends SMSs

Allow **PermAtStartup1** to access this device's location?

This App requests your Location and sends SMSs

Allow **PermAtStartup1** to send and view SMS messages?

(a) Application installation. (b) Granting location pemission. (c) Granting access to SMS. (d) Application info.

Fig. 6 API level 23

occasions, when there is no actual need for using them. In our case, this was made during the first launch of the application. Afterwards, the application was able to access these dangerous resources arbitrarily.

This behavior is rather important for the user. More precisely, the user is constantly being "nagged" to accept a permission, but once accepted it will not be prompted again. The naive user for example would therefore accept the permission permanently, whereas if she was prompted again, even if she had accepted once, she could occasionally revoke this permission. A typical example can be considered in location-aware applications where users would like to selectively disclose their location to the service provider and not to perform a long sequence of actions to revoke such a permission.

Going a step further, one can claim that the permission model can become even more misleading in the cases of the dangerous permission groups. For instance, we discuss the use case where a user installs an application that requires some access to the phone's capabilities. When the app requests to read the phone state (permission `READ_PHONE_STATE`); a widely used permission, the user will be asked to grant access for this request. Nonetheless, after accepting the permission, the user has granted, indirectly, more access to the application, as the application is actually granted full access to the dangerous "phone" permission group. Therefore, the application may read and change the call history (`READ_CALL_LOG`, `WRITE_CALL_LOG`), or even make phone calls (`CALL_PHONE`) without any user notification. Actually, the permission request is stated as "*Allow **ApplicationName** to make and manage phone calls?*". The statement is quite vague, so it may frighten the user, nevertheless, as users are accustomed to such requests most likely they will accept. Nonetheless, the user cannot determine which of the actual permissions has been granted to the application and cannot revoke the permission to a single member of the category.

The confusion of the user may even be greater than the aforementioned notification, as the new permission categorization does not improve transparency. During installation users are not allowed to see what actual permissions are going to be requested by an application in their future executions. More interestingly, since many permissions of Table 1 have not been categorized as dangerous, the user is not prompted about them and they are automatically granted on installation.

An important factor that was overseen in this radical shift is that users would be accustomed to use the new permission model. Therefore, a user is expected to always have the control of dangerous permissions during runtime, regardless of what he has done in the installation procedure. Exploiting this false concept, malware authors have specifically targeted Marshmallow, using the "backwards compatibility" option which allows it to install and execute applications developed for previous API versions. However, in this case, things are not as users would expect. If an application has been developed for `target_sdk` 22, then once a victim installs the application to Marshmallow, it will be granted the permissions upon installation. Definitely, the victim will be shown the "dangerous" permissions upon installation, but since the user is not accustomed to following this procedure during installation (as a result of being used to the new runtime permission model), it is highly likely that she will accept it, hoping that when she launches the application, she will be able to apply her permission policies. This user experience gap has already been exploited by malware, such as `Android.Bankosy` and `Android.Cepsohord`. Although apps do not launch automatically, such apps would exploit this gap to collect as much personal information as possible before the user revokes his permissions.

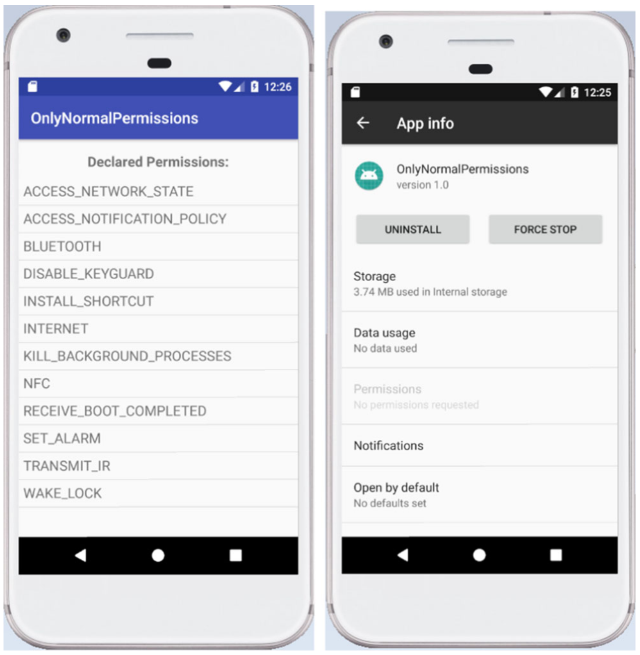
A final consideration concerning the lack of transparency, reported by the authors, is that by requesting only normal permissions, Android apps completely obscure these permission from their users. In the cases where a dangerous permission is included in a set of requested permissions, users are able to navigate to the app's settings and reveal both the dangerous and also the normal permissions that are required by the app. However, after installing an app that has no dangerous permission included in its manifest, users have no access to the underlying permissions through the app's settings menu, since this capability is surprisingly disabled. This can be considered as a very important security issue, not only because of its "lack of transparency" dimension, but also since its exploitation can lead to obvious app metamorphoses attacks. In this case, apps could be initially installed without any permission requirements and would subsequently acquire an arbitrary number of normal permissions in automatic updates, leaving users with complete lack of control over them. One could argue that since these permission are not marked as dangerous, the impact could be rather small. However, this is not the case since normal permissions have been proven to conceal security threats [4, 5, 36], but also because our research has revealed that even "System" permissions are mistakenly handled by the system behaving as normal. The `SYSTEM_ALERT_WINDOW` permission, is considered by Google, as a "System" permission that: "*Very few apps should use this permission; these windows are intended for system-level interaction with the user.*" [7]. Nonetheless, our research has revealed that not only it is automatically granted for all apps that come from the Google Play store, without user interaction, but also, in the case of being included in the "normal permission set," described above, it is also hidden by the users. The security issue in question has also been responsibly disclosed to Google, over a year before the time of writing this paper. Regrettably, the Android team has responded by claiming that this issue "falls more under the category of a feature request as opposed to a security vulnerability," and that "they will pass it along to the feature team for consideration in a future version of Android."

4.6.2 Permission Intransparency

Another issue reported by the authors refers to a lack of transparency in the permission handling mechanism, which has been applied after Android Marshmallow. More specifically, when an app has both dangerous and normal permissions declared, users are able to see both of these categories through appropriate navigation in the app's settings menu (App Info→Permissions→App Permissions→All permissions). Even though the total of declared permission is not illustrated in the first-user interface, perhaps for user experience reasons, nevertheless a user has actually the potential to find them through the settings menu. However, our independent research revealed that if the app has no dangerous permission declared, the "permission" option is disabled, not giving users the opportunity to see and/or review all the apps permissions. The same case applies even in the case of a "system" permissions, namely the `SYSTEM_ALERT_WINDOW` one, where no access from this specific menu is enabled. Transparency in users' permissions means clearly that users should be able to see all permissions requested by apps, no matter whether they are marked as normal or dangerous. If this was not the case in question, then there should be no need in having "normal" permissions at all.

The described use case is illustrated in Fig. 7, where an application clearly declares a dozen of normal permissions, notwithstanding the users have no means to get informed about them through the OSes app settings.

Even more alarmingly, such a security bug also enables mobile apps' metamorphoses attacks. More specifically, apps could initially declare no permission at all, and after


12:26
OnlyNormalPermissions
Declared Permissions:
ACCESS_NETWORK_STATE
ACCESS_NOTIFICATION_POLICY
BLUETOOTH
DISABLE_KEYGUARD
INSTALL_SHORTCUT
INTERNET
KILL_BACKGROUND_PROCESSES
NFC
RECEIVE_BOOT_COMPLETED
SET_ALARM
TRANSMIT_IR
WAKE_LOCK

12:25
App info
OnlyNormalPermissions
version 1.0
UNINSTALL
FORCE STOP
Storage
3.74 MB used in Internal storage
Data usage
No data used
Permissions
No permissions requested
Notifications
Open by default
No defaults set

(a) Normal permissions declared (b) No permissions shown in settings

Fig. 7 Lack of transparency in normal permissions

installation, through app updates they could request an arbitrary number of normal permissions (correspondingly, or even system ones). Afterwards, through automatic updates, they would be granted these permissions and the users would have no knowledge about it, nor a way to get informed, staying completely unprotected in terms of permissions' transparency.

This issue has been found present in all Android versions from Marshmallow and above, tested up to Android Oreo, and has been responsibly disclosed to the involved company.

4.7 Access to the Internet

Clearly, the `INTERNET` permission, as its name suggests, allows an application to connect to the Internet. Up to API 22, Google considered `INTERNET` permission a dangerous one, however, since Marshmallow this is not the case. It is considered a normal one so the user is not notified about it during installation nor afterwards. Notably, due to the Android security model, the user cannot block an application from accessing a domain or the Internet, and additionally, she will not be notified of such actions. One of the core ideas behind this change is the fact that many applications were requesting this permission. As highlighted in [22, 27], while the `INTERNET` permission is widely used and in many cases, it is used only to fetch advertisements [10], yet it is often used to leak private user information, such as location. Google considers that since in Marshmallow there is an inherent mechanism to control access to sensitive pieces of information, an application cannot leak important data about the user without his consent, that is grant access to dangerous permissions.

To illustrate the changes in this particular permission, since the runtime permission model we have used Tacyt[7]. More specifically, we have noticed that there is a significant change in the usage of the Internet permission. Since Tacyt reports the results according to app versions and not per app, in what follows the reported figures refer to versions. Up to the release of Marshmallow, 89.24% of the uploaded versions were using the Internet permission. With the introduction of Marshmallow, this percentage has been increased by more than 7%, reaching 96.26%, indicating that many apps took advantage of this change to allow themselves to have access to the Internet.

Nonetheless, the very existence of a channel that enables an application to connect to the outer world through the Internet, without the user's consent or control essentially augments many security and privacy issues. The reason is that several "benign" actions do not imply any risk for the user, however, if someone can control them remotely or get a result out of them, can greatly expose the user. A typical example of this problem is the clipboard, used by every user to transfer information between applications. Clearly, due to the physical constraints for data input, most mobile users use this functionality to copy passwords, links, or other content from one application to another. Apparently, the sensitivity of the content that is temporarily stored can easily be used to launch an attack [24, 48]. Since there is no special permission for accessing the content of the clipboard, any application can sniff it and transmit it to a predefined location or modify it (e.g., injecting a malicious link). Clearly, this risk could be avoided if the applications had no Internet access or the user could define Internet access policies.

Apparently, the existence of such a channel, facilitates the leakage of other sensitive information. Another example is the access, without requesting any permission, e.g., to local storage, to the drawable area of the wallpaper (reported by the authors, triaged and awaiting for a bug fix for more than a year before the time of writing). While drawing on a canvas cannot be considered harmful, one has to consider that most users use personal photos as their wallpapers, which may depict their beloved, express their political or religious beliefs. Allowing an application to collect this information without user explicit consent could allow it to extract sensitive information, which apart from the aforementioned could include music and sexual preferences, relationship status, etc. Clearly, if most users knew that this information could be mined and processed from apps without their explicit consent, they would be quite reluctant to use many of the photos that they currently do. Moreover, as shown in [5], this can be exploited to leak the user's unlock PIN or pattern.

It should also be highlighted that if users were able to block Internet access per application, it is most certain that in many instances they would so. The reason is that most applications only need Internet access to display ads which for the vast majority of them is the only monetization source.[8] Apparently, if apps are not able to display ads, developers will have to result in other means for monetization in order to support their apps, e.g., shifting to the traditional paid model, which would radically change the Play store, as well as Google's monetization policy from Android. We argue that the answer to this question is not obvious, and there are several ways to avoid this dilemma, e.g., by providing unrestricted Internet access to applications only to fetch ads.

[7] https://www.elevenpaths.com/technology/tacyt/index.html

[8] According to AppBrain(http://www.appbrain.com/stats/free-and-paid-android-applications) the ratio of free to paid apps is more than 10 at the time of writing. Free apps with in app purchases are considered free.

4.8 User Profiling

While Android has been introducing many restrictions to unique identifiers, e.g., since Nougat most content of `/proc` has become inaccessible by apps, there are many ways in which apps using only normal permissions can profile the users.

The `ACCESS_WIFI_STATE`, as well as `CHANGE_NETWORK_STATE` and `CHANGE_WIFI_STATE` permissions have been removed from being declared as "dangerous" in the new permission categorization. Automatically granting these permissions allow an application to connect and disconnect from a WiFi. More interestingly, it allows the application to retrieve the information of the connected WiFi which can expose a lot of information [1]. On a first level of a nefarious scenario, this could allow an application to enforce extra charges to the user by disconnecting from the WiFi and using a 3G/4G connection. Nonetheless, going a step further, one could determine the location of the user from the name of the connected WiFi, but more interestingly, the application can create a user profile as it has access to all stored networks. Apparently, collecting this information, one could correlate it with others to determine social connections using other sources of information such as time to infer, e.g., how long two users stay in proximity, what times of day, etc. harvesting users' relative location and potential relationship.

A lot of usage statics and preferences can be extracted by the apps using the `GET_PACKAGE_SIZE` permission, a normal permission as well. Using this permission, an application can profile a user as it can list all the installed applications and determine the user interests. Additionally, since this permission retrieves the storage space used by an application, an adversary could also infer how much an application is being used. Interestingly, the permission `KILL_BACKGROUND_PROCESSES` is also a normal permission, allowing it to kill other process, apart from system ones. Essentially, this permission can be used to sabotage other applications as they could shutdown, losing needed information or without notifying the user about e.g., an important event.

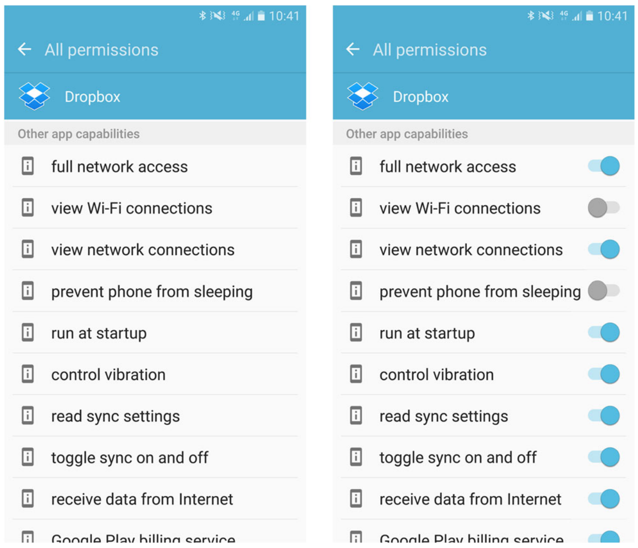
Finally, despite the upcoming changes to Android ID from Android O, apps can use a non user-resettable identifier which is app metadata. In this regard, an adversary can keep track of when apps were installed by reading the metadata of the `/data` directory which constitute a unique identifier [6].

5 Remedies

Perhaps, the most obvious change that is probably needed in the post Marshmallow era is to allow users to have full access to the underlying permissions. This would allow them to both review the permissions that they grant to each application; improving transparency, but also to revoke access on both normal permissions as well as to categories from dangerous permission groups. This does not essentially have to confuse the user since for instance Android would automatically grant normal permissions, but request permission for each dangerous permission and not for a group. Such an interface could be conceived as the right hand side in Fig. 8, which showcases our approach in comparison with the current one. The user can easily see what are the granted permissions per application and revoke those when deemed necessary.

While the latter does not request many changes in the core of Android, a significant change should be introduced in the runtime permission model. As already discussed, applications request permissions to a resource at an irrelevant point, thus misleading users. While the developers may add an explanatory text of why they request a permission, the fact that the user cannot determine when it is actually used by the application does not create a trust relationship with it. For instance, a user cannot understand when an application needs to send an SMS, that might infer some additional cost. Such functionality might be needed once, e.g. for initialization and authorization of the application and never be used again, or be a part of a functionality that the user never uses. Having granted this permission prior to the actual request, authorizing arbitrarily such actions, can lead to malicious applications exploiting this initial trust.

Another vector of the attacks illustrated in this paper deals with the app updates, as mentioned in the previous section. The profound reason for these attacks becoming



Left	Right
full network access	full network access ●
view Wi-Fi connections	view Wi-Fi connections ○
view network connections	view network connections ●
prevent phone from sleeping	prevent phone from sleeping ○
run at startup	run at startup ●
control vibration	control vibration ●
read sync settings	read sync settings ●
toggle sync on and off	toggle sync on and off ●
receive data from Internet	receive data from Internet ●
Google Play billing service	Google Play billing service ●

Fig. 8 Proposed interface for managing normal and dangerous permissions

possible is because of a lack in the control mechanisms from Google Play and Android itself. As discussed, any app can claim arbitrary names, even after installation, and can use resources of other applications. Moreover, the user is not able to determine which the foreground application is, and the updates do not inform the user of changes in permissions names. Therefore, the platform in question does not provide at any point the needed transparency to prevent privilege escalation attackes and also phishing attacks. Furthermore, the malicious apps are able to succeed in their attacks by using other companies' resources, such as app icons, in every case without having the companies' consent to do so. A remedy to this problem would be to use other unique ids in the runtime permission mechanisms, such as app package names, which by their definition in Google Play are unique. Of course, this countermeasure should be accompanied by corresponding permission changes in the way that apps deal with other apps' resources.

Going a step further, even normal permissions can lead to user profiling and expose sensitive user information, as outlined in the previous section. Therefore, Google has to consider what an application can infer from combinations of such permissions through communication between cooperating applications and alert the user of possible consequences.

6 Conclusions

The Android security for API levels prior to Marshmallow had several important issues that should be addressed, something that became apparent after many attacks in core libraries [20, 39] and functionalities that had to be introduced to counter other attacks, such as storage encryption or even setting the device to charge-only mode by default when connected to USB. Google's Android team, made a reasonable decision towards an implementation that would offer more and better protection to Android smartphone users of all ages and backgrounds. A good example is the presence of permission settings in the application manager, where users have real-time access to specific groups of dangerous permissions, after an application is installed. However, major changes in software models that existed for several years can be easily accompanied by new security issues that may arise and new situations where end users may turn out not to be satisfied, nor actually protected.

This paper focused on highlighting several security issues that have risen since the introduction of the "Runtime Permission" model in the Android OS. Additionally, in this paper, we have disclosed a significant number of security issues regarding the Android permission architecture, which have been reported by the authors to Google. Some of the security issues may be addressed by re-organizing permission groups, and perhaps endorse a more "strict" permission policy in terms of "dangerous" and "normal" (not dangerous) permissions.

Other issues regarding user notifications and especially what information is given to users are also quite important. Moreover, we detailed security and privacy issues which are either introduced or augmented by the new permission model. Undoubtedly, a core issue in the majority of the presented apps in this research is the unrestricted Internet access, which provides applications with a communication channel that cannot be stopped or even filtered. As discussed, blocking this path may result to other side effects, which could potentially radically affect and change the Android market. Nevertheless, we believe that Android needs to incorporate more precise explanations to users about permissions, as well as the ability to inform users in more detail. Additionally, not negatively affecting user experience, the OS should incorporate "advanced" settings in security options, where users could, whenever deemed necessary, monitor third party app "behavior" in detail and also manipulate all kinds of permissions.

Finally, the runtime permission model should handle more sufficiently the need to inform users when a dangerous or even a system permission is required, by means of exact time and purpose. At its current implementation, the proposed security model in Android leaves "more freedom" to Android developers to ask for permissions even when they do not need them and keep them for future uses. This combined with the "click once but permanent acceptance" of dangerous permissions can lead to the destruction of what is meant as runtime permission and even more importantly what users expect by it. As already connoted, requesting all app permissions during the first app usage does not necessarily imply any significant differentiation than requesting the permissions during the app installation. We argue that a possible solution to this problem would be to stop expecting developers to do the checks and request permissions in their programs but to force this operation to be controlled by the OS. Correspondingly, this could be accomplished by properly reconstructing Android's programming classes and interfaces involved in all kinds of "dangerous" permissions so as to automatically request user permissions, or check whether a permission has been applied, every time a dangerous resource is handled in code.

Smartphones have radically penetrated in peoples daily lives and accompany them everywhere, at anytime and most importantly being almost always "powered on." This means that a wealth of user data can be collected and used by an increasing number of smartphone applications, which in many cases may compromise users' privacy and security for reasons ranging from data harvesting and data mining to more illegitimate purposes. At the current state of smartphone OSes, users' rights are tightly coupled with applications' permissions, which control each app's access to the smartphone's resources. This paper has focused on providing a thorough investigation and understanding of this significant smartphone software mechanism, as well as presenting and disclosing security flaws that should be addressed in the days to come.

Funding Information This work was supported by the European Commission under the Horizon 2020 Programme (H2020), as part of the *OPERANDO* project (Grant Agreement no. 653704). The authors would like to thank *ElevenPaths* for their valuable feedback and granting them access to Tacyt.

References

1. Achara JP, Cunche M, Roca V, Francillon A (2014) Wifileaks: underestimated privacy implications of the access_wifi_state android permission. In: Proceedings of the 2014 ACM conference on security and privacy in wireless & mobile networks, ACM, pp 231–236
2. Alepis E, Patsakis C (2017) Hey doc, is this normal?: exploring android permissions in the post marshmallow era. In: Ali SS, Danger J, Eisenbarth T (eds) Security, privacy, and applied cryptography engineering - 7th international conference, SPACE 2017, Goa, India, december 13-17, 2017, proceedings. Lecture notes in computer science, vol 10662. Springer, pp 53–73
3. Alepis E, Patsakis C (2017) Monkey says, monkey does: security and privacy on voice assistants. IEEE Access
4. Alepis E, Patsakis C (2017) There's wally! location tracking in android without permissions. In: Proceedings of the 3rd international conference on information systems security and privacy - Volume 1: ICISSP, INSTICC. ScitePress, pp 278–284. https://doi.org/10.5220/0006125502780284
5. Alepis E, Patsakis C (2017) Trapped by the ui: the android case. In: Proceedings of the 20th international symposium on research in attacks, intrusions and defenses. Springer. (To appear)
6. Alepis E, Patsakis C (2018) Session fingerprinting in android via web-to-app intercommunication. Security and Communication Networks 2018:7352030:1–7352030:13. https://doi.org/10.1155/2018/7352030
7. Android Developer Manifest.permission – SYSTEM_ALERT_WINDOW. https://developer.android.com/reference/android/Manifest.permission.html#SYSTEM_ALERT_WINDOW, date retrieved: 28/03/2017
8. Android Source Code (2017) platform_frameworks_base/core/res/AndroidManifest.xml. https://github.com/Android/platform_frameworks_base/blob/master/core/res/AndroidManifest.xml
9. Balebako R, Jung J, Lu W, Cranor LF, Nguyen C (2013) Little brothers watching you: raising awareness of data leaks on smartphones. In: Proceedings of the ninth symposium on usable privacy and security. ACM, p 12
10. Barrera D, Kayacik HG, van Oorschot PC, Somayaji A (2010) A methodology for empirical analysis of permission-based security models and its application to android. In: Proceedings of the 17th ACM conference on computer and communications security, ACM, pp 73–84
11. Bartel A, Klein J, Le Traon Y, Monperrus M (2012) Automatically securing permission-based software by reducing the attack surface: an application to android. In: Proceedings of the 27th IEEE/ACM international conference on automated software engineering, ACM, pp 274–277
12. Blasco J, Chen TM (2017) Automated generation of colluding apps for experimental research. Journal of Computer Virology and Hacking Techniques 1–12
13. Book T, Pridgen A, Wallach DS (2013) Longitudinal analysis of android ad library permissions. arXiv:1303.0857
14. Book T, Wallach DS (2013) A case of collusion: a study of the interface between ad libraries and their apps. In: Proceedings of the third ACM workshop on security and privacy in smartphones & mobile devices, ACM, pp 79–86
15. Calciati P, Kuznetsov K, Bai X, Gorla A (2018) What did really change with the new release of the app? In: Proceedings of the 15th international conference on mining software repositories, ACM, pp 142–152
16. Chen QA, Qian Z, Mao ZM (2014) Peeking into your app without actually seeing it: UI state inference and novel android attacks. In: 23rd USENIX security symposium (USENIX security 14). USENIX Association, San Diego, pp 1037–1052
17. Davi L, Dmitrienko A, Sadeghi AR, Winandy M (2011) Privilege escalation attacks on android. In: Information security. Springer, pp 346–360
18. Diao W, Liu X, Zhou Z, Zhang K (2014) Your voice assistant is mine: how to abuse speakers to steal information and control your phone. In: Proceedings of the 4th ACM workshop on security and privacy in smartphones & mobile devices, ACM, pp 63–74
19. Dimitriadis A, Efraimidis PS, Katos V (2016) Malevolent app pairs: an android permission overpassing scheme. In: Proceedings of the ACM international conference on computing frontiers, ACM, pp 431–436
20. Durumeric Z, Kasten J, Adrian D, Halderman JA, Bailey M, Li F, Weaver N, Amann J, Beekman J, Payer M et al (2014) The matter of heartbleed. In: Proceedings of the 2014 conference on internet measurement conference. ACM, pp 475–488
21. Economist T (2017) The world's most valuable resource is no longer oil, but data. https://www.economist.com/leaders/2017/05/06/the-worlds-most-valuable-resource-is-no-longer-oil-but-data/
22. Enck W, Gilbert P, Han S, Tendulkar V, Chun BG, Cox LP, Jung J, McDaniel P, Sheth AN (2014) Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Transactions on Computer Systems (TOCS) 32(2):5
23. EUGDPR (2018) The EU general data protection regulation. https://www.eugdpr.org/
24. Fahl S, Harbach M, Oltrogge M, Muders T, Smith M (2013) Hey, you, get off of my clipboard. In: International conference on financial cryptography and data security. Springer, pp 144–161
25. Faruki P, Bharmal A, Laxmi V, Ganmoor V, Gaur MS, Conti M, Rajarajan M (2015) Android security: a survey of issues, malware penetration, and defenses. IEEE Commun Surv Tutorials 17(2):998–1022
26. Felt AP, Chin E, Hanna S, Song D, Wagner D (2011) Android permissions demystified. In: Proceedings of the 18th ACM conference on computer and communications security. ACM, pp 627–638
27. Felt AP, Greenwood K, Wagner D (2011) The effectiveness of application permissions. In: Proceedings of the 2nd USENIX conference on web application development, pp 7–7
28. Felt AP, Ha E, Egelman S, Haney A, Chin E, Wagner D (2012) Android permissions: user attention, comprehension, and behavior. In: Proceedings of the eighth symposium on usable privacy and security. ACM, p 3
29. Fratantonio Y, Qian C, Chung S, Lee W (2017) Cloak and dagger: from two permissions to complete control of the UI feedback loop. In: Proceedings of the IEEE symposium on security and privacy (Oakland), San Jose, CA

30. Goodin D (2015) Beware of ads that use inaudible sound to link your phone, tv, tablet, and pc http://arstechnica.com/tech-policy/2015/11/beware-of-ads-that-use-inaudible-sound-to-link-your-phone-tv-tablet-and-pc/
31. Goodson S (2015) If you're not paying for it, you become the product https://www.forbes.com/sites/marketshare/2012/03/05/if-youre-not-paying-for-it-you-become-the-product/#3398a05f5d6e
32. Google: Aosp source code for filesystem_config. https://android.googlesource.com/platform/system/core/+/master/libcutils/include/private/android_filesystem_config.h
33. Grace MC, Zhou Y, Wang Z, Jiang X (2012) Systematic detection of capability leaks in stock android smartphones. In: NDSS
34. Jeon J, Micinski KK, Vaughan JA, Fogel A, Reddy N, Foster JS, Millstein T (2012) Dr. android and mr. hide: fine-grained permissions in android applications. In: Proceedings of the second ACM workshop on security and privacy in smartphones and mobile devices. ACM, pp 3–14
35. Kelley PG, Consolvo S, Cranor LF, Jung J, Sadeh N, Wetherall D (2012) A conundrum of permissions: installing applications on an android smartphone. In: Financial cryptography and data security. Springer, pp 68–79
36. Kywe SM, Li Y, Petal K, Grace M (2016) Attacking android smartphone systems without permissions. In: 2016 14th annual conference on privacy, security and trust (PST). IEEE, pp 147–156
37. Orthacker C, Teufl P, Kraxberger S, Lackner G, Gissing M, Marsalek A, Leibetseder J, Prevenhueber O (2012) Android security permissions–can we trust them? In: Security and privacy in mobile information and communication systems. Springer, pp 40–51
38. Patsakis C, Alepis E (2018) Knock-knock: the unbearable lightness of android notifications. In: Proceedings of the 4th international conference on information systems security and privacy, ICISSP 2018, Funchal, Madeira - Portugal, January 22-24, 2018. pp 52–61
39. Peles O, Hay R (2015) One class to rule them all: 0-day deserialization vulnerabilities in android. In: 9th USENIX workshop on offensive technologies (WOOT 15)
40. Poeplau S, Fratantonio Y, Bianchi A, Kruegel C, Vigna G (2014) Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In: 21st annual network and distributed system security symposium, NDSS 2014, San Diego, california, USA, February 23-26, 2014. The Internet Society
41. SnoopWall (2014) Flashlight apps threat assessment report http://www.snoopwall.com/wp-content/uploads/2015/02/Flashlight-Spyware-Report-2014.pdf
42. Taylor VF, Martinovic I (2017) To update or not to update: insights from a two-year study of android app evolution. In: Proceedings of the 2017 ACM on Asia conference on computer and communications security. ASIA CCS '17. ACM, pp 45–57
43. Tsiakos V, Patsakis C (2016) Andropatchapp: taming rogue ads in android. In: Mobile, secure, and programmable networking - first international conference, MSPN 2016
44. Tuncay GS, Demetriou S, Ganju K, Gunter CA (2018) Resolving the predicament of android custom permissions. In: ISOC network and distributed systems security symposium (NDSS)
45. Wei X, Gomez L, Neamtiu I, Faloutsos M (2012) Permission evolution in the android ecosystem. In: Proceedings of the 28th annual computer security applications conference. ACM, pp 31–40
46. Wibson (2018) How much is your data worth? At least $240 per year. likely much more https://medium.com/wibson/how-much-is-your-data-worth-at-least-240-per-year-likely-much-more
47. Yang L, Boushehrinejadmoradi N, Roy P, Ganapathy V, Iftode L (2012) Short paper: enhancing users' comprehension of android permissions. In: Proceedings of the second ACM workshop on security and privacy in smartphones and mobile devices. ACM, pp 21–26
48. Zhang X, Du W (2014) Attacks on android clipboard. In: International conference on detection of intrusions and malware, and vulnerability assessment. Springer, pp 72–91