Trapped by the UI: The Android case

Efthimios Alepis, Constantinos Patsakis

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

Abstract

Mobile devices are highly dependent on the design of user interfaces, since their size and computational cost introduce considerable constraints. UI and UX are interdependent since UX measures the satisfaction of users interacting with digital products. Therefore, both UX and UI are considered as top priorities among major mobile OS platforms. In this work we highlight some pitfalls in the design of Android UI which can greatly expose users and break user trust in the UI by proving how deceiving it can be. To this end, we showcase a series of attacks that exploit side channel information and poor UI choices ranging from sniffing users' input; resurrecting tapjacking, to wiping users' data, in Android from KitKat to Nougat.

1 Introduction

Modern mobile devices have penetrated our everyday life at an unprecedented rate. An indicator of this trend is that despite the fact that commodity smartphones date back to less than a decade, globally there are more smartphone users than desktop users. In terms of capabilities, while they can be considered as a stripped down version of modern computers, their various embedded sensors provide them additionally allowing them to sense their location through *e.g.* the GPS, their position through the compass, or even the motion of a device through accelerometers. This knowledge allows smartphones to adjust the user interface and the provided information in real-time in a way that fits better for the user and the corresponding environment.

More than simply managing all this information in a computational efficient way, mobile devices are subject to size constraints as the attached monitor which acts as both an input and an output modality of interaction is rather small and a lot of functionality has to be squeezed into it in the most intuitive way so as not to confuse users when interacting with the device. As a result, mobile UIs contain a lot of components and information in a rather confined setting. Therefore, while the resulting UI seems rather simple, it is in fact fairly complex. Furthermore, since all mobile applications share the same small screen, they end up getting stacked one on top of the other which prevents users from determining to which application the foreground component belongs to. Nevertheless, users have absolute trust in the UI: they expect that what they are presented is exactly what it claims to be.

Smartphone UIs have received a lot of attention over the last years, with numerous researchers revealing vulnerabilities that lead to a significant number of OS patches and precautionary measures, with Android; due to its popularity, receiving most of them. In this paper, we present new attack vectors that we have discovered which not only bypass recent countermeasures integrated in Android, but more importantly, these attacks, in many cases, are for more malicious than the reported in current state of the art.

After reviewing the related scientific literature, we argue that one may categorise Android UI attacks into two main categories. The first category consists of attacks that utilize window dialogs that hold the `SYSTEM_ALERT_WINDOW` signature level permission, allowing them to be shown on top of all other apps; e.g. [41]. Android Toast messages are an exception in this category, since they require no permission, however have some significant limitations, as it is discussed in the next section. The second set of attacks consists of applications that manage to determine the foreground app and consequently present a fake application to steal sensitive user information ; e.g. [18]. However, at the time of writing, these attacks have either low or zero impact since their underlying security issues have been already addressed. For the first category of attacks, the `SYSTEM_ALERT_WINDOW` permission requires special handling by the user, after the installation of an app to be granted. Additionally, after the introduction of API level 23, special intents (e.g. `ACTION_MANAGE_OVERLAY_PERMISSION` [11]) and checks (e.g. `canDrawOverlays()` [11]) have been introduced to harden the UI and disable third party apps from arbitrarily drawing over other apps. Regarding the second set of attacks, again several countermeasures have been applied during the last years. Moreover, Android ActivityManagers class method getRunningTasks(), has been deprecated in API level 21 and is no longer available to third party applications [2], while ActivityManager's class `getRunningAppProcesses()` returns a list of only the caller applications package name as of API level 22.

In our work we use quite different attack vectors to achieve these results, which, to the best of our knowledge and according to our reports to Google; see Table 1 for details, had not been studied yet. Our proposed attacks exploit some of the properties of the most generic Android OS mechanisms, such as Android activities and Intents. Hence, not only do we succeed in delivering a wide range of attacks to the Android OS through seemingly benign apps; they do not request any dangerous permission, but we also provide proofs that these vulnerabilities exist for far too many years, up to the latest versions. These attacks may range from stealing sensitive input and installing apps without users' knowledge, to wiping the user's phone, even in the latest versions of Android AOSP (SDK 25). In addition, we have successfully uploaded our proof of concept applications to Google Play, bypassing the security checks from the Bouncer; the system which analyses applications in Google Play for malicious functionality [27, 31], further proving the significance of the threats.

While much effort has been made in Android towards countering UI redressing attacks, for instance since Marshmallow, the user is presented with a notification screen whenever an overlay is detected, the ground truth is that most of these defense mechanisms have been partially deployed, allowing an adversary to launch a wide set of attacks. Table 1a provides an overview of our contributions stating some of the Android's design goals and linking them with both our findings and the way that these findings can be used maliciously.

More precisely, in this work we demonstrate that many security standards of Android's UI can be easily bypassed with the use of inherent mechanisms that do not require any special permission from the user. To this end, our presented attacks either exploit the knowledge of what the foreground app is (SDK<24), or lure the user to use arbitrary UIs and result in a series of "unwanted" actions. Based on the methods that will be presented, an adversary can launch several serious attacks, ranging from sniffing sensitive and private data, to gaining administrative privileges that allow the adversary to reset the device, wipe user's data, or even cover the installation of new downloaded apps. A summary of the attacks presented in this work, their applicability to specific Android versions and the percentage of current devices affected by them, are illustrated in Table 1b. It should be noted that the reported results have not been tested to API levels below 19 as these devices not only represent a small market share, but they have been long deprecated.

2 Related work

Android User Interfaces take place in three-dimensional space, where the two dimensions control the horizontal and vertical positioning of controls inside a mobile window respectively, while the other controls the "depth". The latter dimension refers to the different "layers" of UI graphic elements which are placed on a mobile screen and it is defined from the level of the screen towards the user's eyes. Hence, as far as activities are concerned, the "outermost" activity is practically the active one. However, there are also other types of graphic elements that may appear on a mobile screen, such as dialogs. Dialogs consist of controls that may appear on top of activities to interact with the users, usually providing some kind of information. For managing all the UI elements on the Z axis, there is a dedicated Android interface, namely *WindowManager* [12], used by the apps to bound to a particular display.

In terms of user interaction, Android's activities and dialog windows have significant differences between them. First and foremost, activities have a much more complex lifecycle which consists of special states and their corresponding events that are triggered during their lifecycle. On the contrary, dialogs are usually either informative or prompt users for making a decision, and therefore have much shorter and less complex lifecycle. Furthermore, all activities have to be declared inside the app manifest file, whereas there is no such requirement for dialog windows. Following the same logic, an app's activity can also be launched from other apps using *intents*, a special Android mechanism to enable a kind of "communication" between applications through asynchronous messages. Consequently, there is a strong relationship between activities and intra- and inter-app

Design goal	Findings	Issue	Malicious usage
Apps should not access users' private files without specific dangerous permissions (*e.g.* storage) [6]	Apps, without any permission, have read access to the contents of the device's Wallpaper	219663 224516	Using the Wallpaper image we create a fake pin/pattern screen and sniff users' pins and patterns.
Apps should not be aware of users' current foreground app [2].	All apps in Android versions prior to Nougat have access to `/proc/` file system and can determine the foreground app without requesting any permission.	233504	Having actual knowledge of the foreground app we launched attacks involving the app's UI which resulted in sniffing user input data.
Only apps that hold the signature permission `SYSTEM_ALERT_WINDOW` should be allowed to draw over other apps and on a part of the screen [7].	We achieved this goal by using simple Android's activities and without requiring any permission. This way, arbitrary number of sized activities can be stacked over other apps.	233504 234399	We draw over very special OS UI elements involving systems settings, managing to use system apps (phone calls, sms, etc) and third party apps to either complete a malicious action or sniff personal data. We also manage to install new apps.
Every app is able to retrieve a list of all of the device's installed applications [10].	Works as intended.		Having actual knowledge of the installed applications, an adversary can create fake UIs that mimic the real applications. To convince the user in launching the fake UI, the adversary can issue fake notifications or create fake shortcuts.
Even when a UI element is drawn over another app, in critical cases all the background interactions must be either disabled or notify the user. This mechanism has been activated for all dangerous runtime permissions since SDK 23.	While Android successfully detects overlays in dangerous permission settings, it fails to do detect them in signature permissions, such as `SYSTEM_ALERT_WINDOW`, `BIND_ADMIN`, etc, or even when using Package manager.	233504	We can escalate privileges of zero permission apps to gain signature permissions for them (*e.g.* admin rights, change device settings, system alerts) or to download and install new apps.
Each application should have its own notifications so that the user could know its source.	As of API level 23 the developer can provide arbitrary icons for the notification and either provide an arbitrary title (API<24) or strip the application's name (API>23).	233790	An adversary can push malicious notifications to the user, tricking him into launching forged activities that mimic the look and feel of installed legitimate apps.
Applications are able to create shortcuts to facilitate users in launching them. For security reasons, the icon and the corresponding label must be unique and uniquely identifiable by the user	As of API level 19, apps holding the INSTALL_SHORTCUT permission are allowed to create shortcuts to the home screen with arbitrary icons and labels.	234044	A malicious app creates a fake shortcut of a legitimate installed app to the user's home screen, tricking the user in interacting with a malicious app instead of the legitimate one and resulting in private data leakage.

(a) Overview of main contributions and comparison with design goals in Android.

Android Version	Nougat	Marshmallow	Lollipop	KitKat	Available in Google Play	Vulnerable devices(%)
Tested API level	25	23	22	19		
Sniff lock pin/pattern	✓	✓	✓	✓	✓	86.3
Sniff data from foreground apps		✓	✓	✓	✓	85.6
Unauthorized actions through OS apps	✓	✓	✓	✓	✓	86.3
Gain escalated privileges	✓	✓	✓*	✓*	✓	86.3
Interfere UI of legitimate apps	✓	✓	✓	✓	✓	86.3
Fake apps mimicking legitimate apps	✓	✓	✓	✓	✓	86.3
Forged notifications	✓**	✓			✓	30.3
Forged shortcuts	✓	✓	✓	✓	✓	30.3
Install applications	✓	✓	✓	✓	✓	86.3
Revised tapjacking	✓	✓	✓	✓	✓	86.3
Market share (%)	0.7	29.6	33.4	22.6		

(b) Attacks and applicability. *: On API>22 more signature runtime permissions are available *e.g.* `SYSTEM_ALERT_WINDOW` and `WRITE_SETTINGS`. **: On API> 23 the title of the notification is fetced from the app title. Nonetheless, one could declare a name with many spaces or dashes to trick the user.

Table 1: Summary of our attacks and their applicability.

navigation, which makes them one of the fundamental building blocks of apps on the Android platform.

At this point it is essential to clarify how Android UI elements interact with each other inside or outside the scope of an app. In principle, a dialog window cannot appear outside the scope of its calling app; *i.e.* appear on top of another app's activity, unless this app is granted the `SYSTEM_ALERT_WINDOW` permission. While there is an exception to this rule inside Android, concerning the "Toast" window, this type of windows have limited functionality and very short lifetime (maximum 3.5 sec). Moreover, the `SYSTEM_ALERT_WINDOW` permission is a signature level permission; far more strict than dangerous permissions, and allows an application to create windows shown on top of all other apps by using the `TYPE_SYSTEM_ALERT` option. According to Google Developer resources [7]: "*Very few apps should use this permission; these windows are intended for system-level interaction with the user*". Apparently, while many applications may request this permission, this permission is actually neither automatically granted nor the user is notified about it during installation. Therefore, a permission management screen is presented to the user to grant it [7] and to allow the application to draw on top of the others. Table 2 provides an overview of the properties of all UI elements that are able to draw over other apps have.

On the contrary, a newly launched activity is by default, and without requiring any permission, able to appear on top of another app. This is the usual and obvious way of interaction inside Android OS where apps appear on top of others, usually as a result of users' actions, creating a kind of an application stack. Activities which are launching other activities or other apps' activities and sometimes even return results, are a commonplace in Android and are thus thoroughly supported through the Android Intent [5] mechanism. Notably, up to recently, each application would have been actually stacked on top of the others covering them entirely, as the size of each application would have been equal to the screen's size. This is not the case any more as several features, recently introduced in Android's UI, are providing more complex stacks such as messaging apps' "chatheads"; through `SYSTEM_ALERT_WINDOW` permission, and Multi-Window [8].

UI Window Type	**Required Permission**	**Manifest Declaration**	**Focusable**	**Duration**	**Launch from service**
Toast Messages		Not required		3.5 sec	
Alert Messages	`SYSTEM_ALERT_WINDOW`	Not required	✓	No limit	
System Alerts	`SYSTEM_ALERT_WINDOW`	Not required	✓	No limit	✓
Keyguards*		Required	✓	No limit	✓
Normal activity		Required	✓	No limit	✓
Transparent Activity		Required	✓	No limit	✓
Small Shaped Activity		Required	✓	No limit	✓
Notification		Not required		No limit	✓

Table 2: Android UI elements over other apps.

2.1 Attacks to the UI

In principle, one of the main goals of malware is to perform unauthorised actions on victims' devices and for achieving this an adversary may use various approaches. Nonetheless, if the adversary cannot find a vulnerability to penetrate into the user's device either remotely or by getting physical access to it, one alternative way would be to trick the user into performing the malicious action himself. To this end, the adversary may use social engineering methods to convince the user to *e.g.* install a malicious application or change specific OS settings. Obviously, the application must not raise an alert to the user indicating its maliciousness, otherwise the user will not perform the task.

However, even if the user is tricked into installing a malicious app, this does not guarantee that the adversary will accomplish his/her goals. For instance, if the adversary has the goal of stealing a user's password, then the embedded security mechanisms of the operating system may prevent the adversary from this theft. To overcome this obstacle, many malicious applications try to trick users into providing the necessary input directly to them. An obvious method to achieve this is by disguising themselves as legitimate apps so as to trick users into providing the input to them. Another approach, which is very often used in mobile devices due to their UI, is to provide a transparent layer on top of the legitimate application and thus to steal the sensitive user input.

In literature, several attacks targeting Android relevant to our work are documented. Despite the fact that transparent elements in browsers were used as the first UI redressing attacks, we are not studying them hereafter since they target an entirely different environment. Besides, these attacks cannot recover the sensitive information nor can perform the actions that we target in our attacks. To the best of our knowledge, the first attempts to escape the browser environment can be attributed to Richardson [32] and Johnson [22]. Nevertheless, these attacks were quite limited as *e.g.* they used a simple toast. The actual successful UI redressing attack can be attributed to Niemietz and Schwenk [30] who ported them to Android. The authors managed to create an overlay which is "touch transitive" in that the clicks are also transferred to the application which is positioned below it. In that scenario for example, the user is tricked into clicking at specific points on the screen while his clicks are also parsed to the dialer application which sits below the app. In that way, the user performs an unauthorised call to a premium number without realising it.

The attack of Chen et al. [18] starts from a side channel attack to the underlying GUI system. While in principle the attack can be launched to any GUI, the authors focus on Android and more precisely try to infer the activity that is displayed from the foreground application based on shared memory. First, the authors monitor offline the memory counters (virtual and physical) of an application as they are recorded in `procfs`. That is, they monitor the memory consumption of each activity in an application by tracking the memory allocation of the corresponding `/proc/[pid]/statm` file. The hypothesis is that the transition from one activity to another has a specific memory footprint that can be used to create a unique signature in a target app. Based on this signature, the adversary can infer the foreground application and the corresponding activity. By monitoring network traffic through `/proc/net/tcp6` one can further improve the results. Based on this input, the adversary may determine whether, for instance, the victim is presented with the login screen of a sensitive app or he/she is being asked to enter payment details. Therefore, he can timely bring his malicious app in the foreground with a replicated UI and trick the user into disclosing the sensitive information.

Bianchi et al. in [16] categorise all Android UI attacks under the general umbrella of GUI confusion attacks. Despite the countermeasures that are discussed in this work, of special interest to our work is the reported leakage of foreground application by `profs`. In this case, the leakage is from the file `/proc/[pid]/cgroups` whose contents change from `/apps/bg_non_interactive` to `/apps` when an app is sent to the foreground. Recently, Fernandes et al. [21] showed that one could exploit the use of a defense mechanism such as the aforementioned one, by monitoring the binder IPC calls in `/sys/kernel/debug/binder`. This allows an adversary to know when the scan has finished and to timely present user with a fake activity to steal the sensitive data. To overcome this drawback, Fernandes et al. provide a more advanced mechanism which mitigates such attacks.

The attacks of Ying et al. [41] can be considered quite narrow and the assumptions that the authors make are rather strong. Firstly, the attack is mainly focused on custom ROMs where the ecosystem is very different compared to Android AOSP, as there is a lot of customisation and radically different implementations even for native libraries. Actually, the authors exploit one of these features, more precisely the `SYSTEM_ALERT_WINDOW` permission to draw on top of other windows. Notably, to grant this permission to an application the user has to perform a set of actions post installation [7]. Using Tacyt[1] to estimate the exposure from this attack vector, we identified 235,059 versions in Google Play and 28,533 version outside it which use this persmission. The reason for this choice is that Tacyt downloads all the apps from Google Play and all their versions in daily basis, analyses them and provides an interface to mine part of this information. Due to the implementation of Tacyt, the responses are in per app version and not per app, nonetheless, they provide a very good snapshot of available Android apps. Notably, the latter numbers contradict the reported ones by Ying et al. [41]. For Android AOSP, their attack cannot be considered valid as none of the big corporations which are explicitly granted this permission by Google would try to exploit it as such actions would most probably put them out of business immediately. Alternatively, the user has to be tricked into performing a set of unusual and very dangerous actions.

Currently, there are several reported possible countermeasures to UI redressing attacks [1, 29, 28, 38, 21]. Nonetheless, in terms of state of practice we consider as baseline the latest version of Android AOSP, which at the time of writing is 7.1.1. The reasons for this choice is that while several defense mechanisms are implemented for quite old Android versions *e.g.* Android 4.4 and they actually

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

account for a low percentage of market share which has currently been left unsupported. Additionally, Google has issued several security features in the newer versions to tackle many of these attacks, and introduced new UI features, some of which are exploited by our attacks. Notwithstanding the above, the attacks that we demonstrate here can be launched to a plethora of Android versions, illustrating that the defense mechanisms are rather low. For a more thorough overview of this field, the interested reader may refer to [37].

3 The attacks

The following paragraphs present the backbone of our attacks. After introducing our threat model, we provide the necessary technical details and research findings that enable the realisation of our attacks. Based on these findings, we detail how an adversary can take advantage of them to launch an attack.

3.1 Threat model

Like most attacks on Android, we assume that the victim has been tricked into installing a malicious app [20, 36, 19]. To minimize the risk of alerting the user that the app might be malicious, we minimize the requirement for permissions, by requesting access only to the Internet. The latter is a weak assumption, since after the radical changes in Android 6, the new permission model considers this access as a normal permission. Practically, the user is not notified about it, yet the permission is automatically granted and cannot be revoked. Therefore, our threat model assumes that the device has not been compromised via a root exploit. In fact, as we are going to show, most of our attacks can be applied to the latest version of Android. Therefore, our malicious apps are assumed to be unprivileged and managed to trick Bouncer and be shared through Google Play.

To provide stealthiness to our app, instead of just using Internet to communicate the commands and results, we use Firebase. The idea behind this choice is that Firebase provides a nice hide out for the execution of our attack since the channel is considered secure and trustworthy, as it is powered by Google, and the traffic is also considered legitimate as many applications use it to store information. Additionally, it facilitates the development lifecycle as Android has many native API calls to exchange information with Firebase. In this regard, the use of Firebase can be considered similar to the use of Facebook, Twitter, etc. by social botnets [23] to hide their communication with the C&C server.

3.2 Drawing over other activities

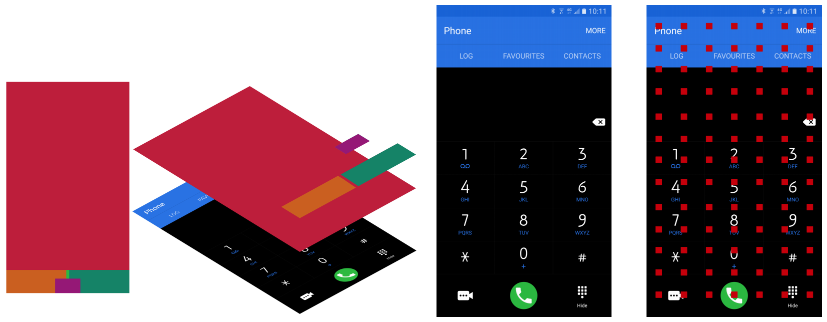
Microphone and touchscreen inputs can be considered as the most sensitive information on a smartphone, as they constitute the primary inputs to the device. While for the former the main security mechanisms can be found in the transport layer, for the latter the mechanisms are embedded in the operating system. This is perhaps the reason why a significant number of corresponding attacks has already been reported. For instance, apart from the obvious keylogger applications, an attacker may try to recover information from leaks (potential or malicious) of the software keyboard [17], processor's cache [24], motion sensors [39, 14], distortions of the wireless signals from finger motions [42], hand motion [26], audio [25], video [33] or both [34] to infer user's input.

In our approach, we exploit Android's UI and side channel information to either steal or interfere with the user's input. In what follows we discuss how one can draw on top of other activities. Practically, this is split in two cases: one where a transparent overlay activity covers another one, and one where one or more non transparent activities partially cover other activities.

For the former case we use typical Android manifest theme declarations. More specifically we have used the `Theme.Translucent.NoTitleBar` parameter in the activities' theme declaration to make an activity transparent and extend it to full screen. In the cases where "on-screen" actions, such as clicks, or key input through the supplementary presence of a keyboard needs to be recorded, the transparent layout was supplied with corresponding `KeyListener` and `ClickListener` objects. Notably, drawing over the Android UI by utilizing a transparent activity was seamless, as a "layer" since any visible view on the transparent activity is seen as visible on the mobile screen (*e.g.* TextViews, Buttons, etc.). The latter case was more demanding as activities whose size is smaller than the screen are statistically quite "rare" in Android apps. Moreover, apart from this constraint, we required to leave user interaction pass through the outer space of the activity. To achieve these, we defined *Application Theme* styles that contained the following items as elements:

```
<item name="android:windowIsFloating">true</item>
<item name="android:windowIsTranslucent">true</item>
<item name="android:windowBackground">@android:color/transparent</item>
<item name="android:windowNoTitle">true</item>
```

Then, a crucial step was to override the activity's `onCreate()` method to define some additional parameters. Namely, a `WindowManager.LayoutParams` object was created whose `dimAmount` was set to 0 and it was flagged with the attributes `FLAG_LAYOUT_NO_LIMITS` and `FLAG_NOT_TOUCH_MODAL`. To position the sized floating activity on the screen, one can fine tune several parameters of the corresponding `LayoutParam` *e.g.* "Gravity" parameters, or actual position through (X,Y) on-screen coordinates. Finally, to make an activity "wrap" around its contents (*e.g.* ImageViews) its `layout_width` and `layout_height` parameters have to be defined to take the `wrap_content` value, instead of the default `match_parent` default value. Notably, the aforementioned properties, can be used to create arbitrary stacks of activities on top of others, allowing an adversary to create an interface as in Figure 1a, where only part of the activity on the bottom can be seen, nonetheless, the interaction (click) is passed to it. The fact that arbitrary number of sized activities can overlay other apps can also be used to create a grid on top of the screen as illustrated in Figure 1b. Both of these approaches, concerning a number of floating, sized activities are used in our research for a wide variety of attacks that range from permission escallation attacks, to revisiting tapjacking, as it is illustrated in the next section.



(a) Stacked overlay activities. (b) Grid from overlay activities.

Fig. 1: Exploiting floating Android activities.

3.3 Tricking users to open apps

In API level 4 Google introduced notifications to Android. As the name suggests, this mechanism notifies users about application events. To create a notification there is no permission needed to be granted. From API level 11, one must denote the text of the notification; through `setContentText` which accepts a string variable, the title of the notification; through `setContentTitle` which also accepts a string variable, and the notification icons for the status bar and the notification view, using `setSmallIcon` and `setLargeIcon` respectively [9]. As of API level 23, both icons can be set dynamically using custom bitmaps. Prior to API level 23, only the `setLargeIcon` provided this feature, as `setSmallIcon` required an integer which denoted the resource ID in the application's package of the drawable to use. Practically, this means that a developer can now fetch all the content of a notification; strings and icons, from the Internet, without having any restriction from the declared app resources in the package. Notably, these attacks emerged since API level 23. While one could long press on the icon of a notification to see its properties, which would actually show the user the correct app, this cannot be considered a normal user interaction, as it beats the perpose of the notifications and cannot be expected to be performed regularly.

Shortcuts are an easy way to launch applications beyond going through the list of installed applications. To this end, they are created in the home screen of Android so that the user can quickly find the apps she uses most often. While the user can create shortcuts for her apps and arrange them in the home screen, applications can also do it when deemed necessary, as long as they have declared the normal permission `INSTALL_SHORTCUT` in their manifest. The underlying mechanism to create a shortcut is intents [4], so the developer has to declare three variables: a string which denotes the caption of the shortcut (`EXTRA_SHORTCUT_NAME`), a string which denotes the "action" of the intent to be launched (`setAction`), and its icon as a bitmap (`EXTRA_SHORTCUT_ICON`). Again, as in the previous case of notifications, all the parameters for the creation of app shortcuts can be set dynamically, using Internet resources.

3.4 Sniffing PIN/pattern

Due to the sensitivity of the data stored in modern smartphones, a wide set of authentication and authorization methods have been introduced to prevent unauthorised access. Perhaps the most common mechanism, regardless of the underlying platform, is the lock screen, where users have to enter a PIN or pattern to unlock the device. The approach is rather simple and provides baseline security from physical intrusion. According to a recent study [35], most users lock their phones by preferring patterns over text based methods; PIN and passphrase.

While in some versions there might be some minor filename changers, by default, the pattern is stored as an unsalted SHA-1 hash in `/data/system/gesture.key` file, while the PIN or passphrase are stored in `/data/system/password.key` as a concatenation of the password's SHA-1 and MD5 hash values. Contrary to the patterns, the text-based passwords use a salt which is stored in the `/data/system/locksettings.db`. Clearly, due to the location where these files are stored, users and applications cannot access them neither for reading nor for writing them. Therefore, attacks to recover the unlocking code are focused either to cases where one has access to the storage and manipulates it to *e.g.* remove the protection mechanism or to sniff the password by side channel attacks [13, 42, 40].

While the user is not allowed to read nor modify the content of the two aforementioned files, an application is able to determine which is the locking modality that is used. To achieve it, the application must simply request the file size of the two files. Obviously, the file whose size is a positive number indicates which of the two modalities is used, as both files exist in the filesystem regardless of which modality the user prefers.

To replicate the lock screen's UI, one also needs to collect the user's wallpaper. Notably, in Android, all applications are allowed to access device's wallpaper by requesting the `getDrawable` property without the need for declaring any dangerous permission, as reported by the Authors, in Security Issue 219663. This choice can be considered rather dubious as users would most often use personal photos as their wallpaper. Clearly, apart from the use described in our attack, this feature also enables apps to profile users since the content of the wallpaper could reveal social connections, religious and political beliefs or even sexual preferences.

Combining the above information we are able to prepare the screen that is be presented to the user when he wants to unlock the phone, since the device's secure lock background image is almost always the blurred version of the user's wallpaper. The malicious application is seemingly harmless and can consist of several activities. Obviously, the fake lock screen functions as the real one, yet it records all touch events, which are stored and transmitted to the adversary to recover the unlocking code. The interface and steps of our attack for the case of pattern locked smartphone are illustrated in Figures 2a,2b and **??**.

To accomplish an attack that will result in sniffing a user's lock screen pin or pattern, our approach requires the implementation of a BroadcastReceiver class that will be capable of listening for screen-off events, (`ACTION_SCREEN_OFF`), while our app is running on the foreground. In other words, the actual initialization of our attack is triggered by the user, not when she tries to unlock her mobile phone by using the power button, but on the contrary when she locks her phone so that she will subsequently unlock it for the next use. As a result, our fake lock screen will be brought to the foreground after the screen-off event and will remain there invisible until the moment the user tries to unlock her smartphone. However, due to Android OS's restrictions for security reasons, this "special" kind of broadcast receiver cannot be registered in the app's manifest but only programmatically on runtime, nor can it be associated with a different activity than the one that registered the receiver. To overcome these restrictions our app registers the broadcast receiver programmatically through a "dummy" activity and most importantly the same activity is also used to create the fake lock screen. We accomplish this "transformation" of the dummy activity into the desired one by hiding all the views that were used in it and by replacing them with visible ones that where previously hidden, which comprise the "desired" fake lock screen activity. Of course, the device's specs are "welcomed" by attackers in order to "fine tune" the attack, such as screen size and screen fonts, and thus produce a "convincing" result. In order to force our fake lock screen precede the real lock screen when the victim presses the power button, some special flags are used, such as the `FLAG_SHOW_WHEN_LOCKED` parameter. Finally, while the user interacts with our fake lock screen we manage to create a simple path data structure where each (X,Y) coordinate regarding touch screen events and movements is recorded, `ACTION_DOWN`, `ACTION_UP` and `ACTION_MOVE`. Obviously, analysing this data structure can straightforwardly reveal a victim's lock screen pattern. Certainly, producing a fake lock screen that consists of UI controls to capture a 4 digit screen lock is simpler.

3.5 Inferring foreground application

For obvious security and privacy reasons, Android prevents applications from inferring which application is on the foreground. Nonetheless, it allows applications to know, without requesting any dangerous permission, which applications are installed in the device, as well as which ones are currently running; the latter only applies for all Android versions prior to Nougat. While these permissions and restrictions are performed in the SDK, one may dig into the OS layer to retrieve this information.

Android is practically a Linux system and as most of the Unix-like systems it follows the same approach for handling its filesystems. One well-known, yet special filesystem is `procfs` which is utilised to store the information of the processes that are executed by the operating system. While accessing the information in this filesystem is well protected, in terms of reading and altering the stored information this does not actually prevent side leakages. In principle, in Android these mechanisms are more strict as each application is a separate user, and as such, each application is prevented from accessing the "internals" of the other. Nonetheless, some metadata are publicly available to all applications.

Special concern should be paid to the `oom_adj_score` file. To understand the importance of this parameter we will discuss some Android specific features of process management. In principle, Android runs is mobile devices which have constrained resources, whereas many refinements have been introduced by Google in order to allow Android to perform resource allocation and release. Since the device has limited memory, Android performs the following steps to achieve stability. If there is memory free, Android uses Zygote to launch a new VM. However, if there is not any free memory, it has to close the last user application. In this regard, each application is given a `oom_adj_score`, stored under `/proc/[pid]/`. By monitoring the aforementioned files, and pruning all the system applications, one can easily determine which is the application which is less probable to be killed, which eventually, is the foreground app.

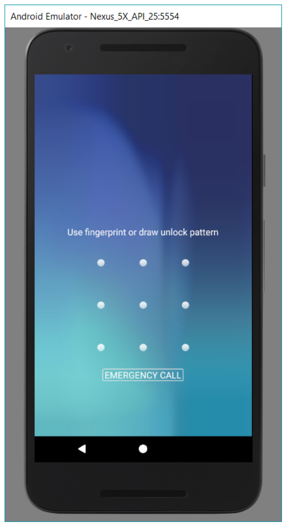
4 Use cases and implemented attacks

To demonstrate our attacks and highlight their importance, we have prepared a set of different attack scenarios that reveal different exposures from the Android UI. Some representative interfaces of the attacks that we launched are illustrated in Figure 2. In these screenshots we have deliberately created a sloppy interface for most of the attacks so that the reader can easily determine the overlayed activity as well as the exposed functionality. As discussed in the previous section, an adversary can easily present either transparent or sized activities on top of the benign ones to provide the necessary look and feel and trick the user into performing illegal actions and/or sniff input data.

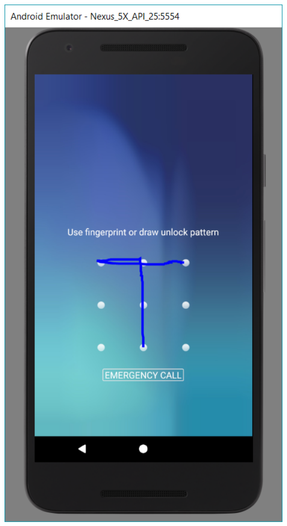
While one could argue that the activities and their resources must be declared in the manifest, one can easily bypass this restriction by simply using webviews that cover the whole activity. In this regard, an adversary can load dynamically any interface he wants. Note that the adversary through his malicious app already knows which apps are installed in the victim's device, also illustrated in Figure 2, so he can easily prepare the appropriate interface and load it dynamically when deemed necessary. Therefore, in what follows, we consider the creation and delivery of the forged interface as a trivial part of the attack that is made mostly through Firebase.

The lifecycle of our attacks is the following. Initially, Malory, the adversary, uploads the malicious app in Google Play; as already reported our apps bypass Bouncer's filters, and the user is tricked into downloading the app and installing it since it requires no special permissions. Then, the app sends through Firebase all the necessary input from the victim's phone. Next, Malory delivers all her payload for the attacks through Firebase. Depending on the installed apps and Android version, the malicious app either timely launches a forged activity or overlays a benign app.

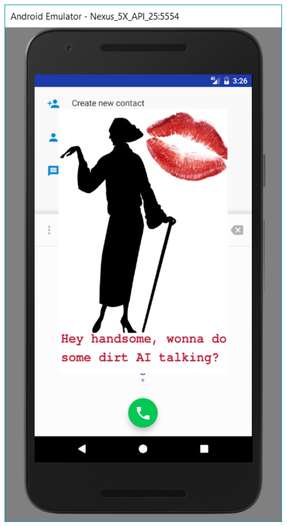
Starting a phone call: While an application needs to have a dangerous permission granted to start a call, any application can use an intent to launch the "Phone" application with an arbitrary number to call. For obvious reasons this call will not be made unless the user presses the call button. Exploiting the UI


Android Emulator - Nexus_5X_API_25:5554
Use fingerprint or draw unlock pattern
EMERGENCY CALL

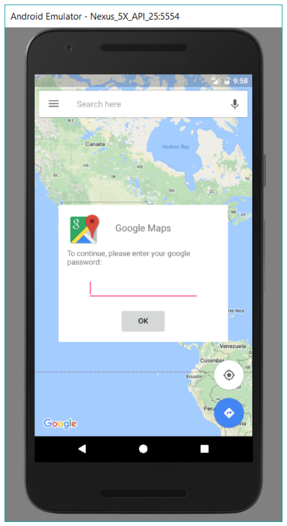
(a) Forged lock screen.


Android Emulator - Nexus_5X_API_25:5554
Use fingerprint or draw unlock pattern
EMERGENCY CALL

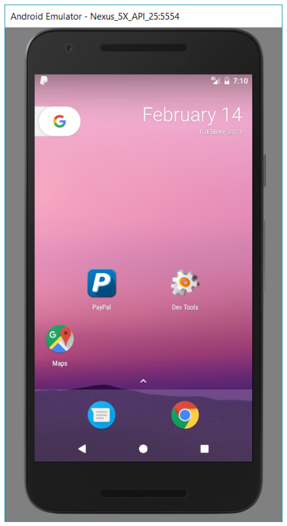
(b) Recording the user's pattern.


Android Emulator - Nexus_5X_API_25:5554
Create new contact
Hey handsome, wonna do some dirt AI talking?

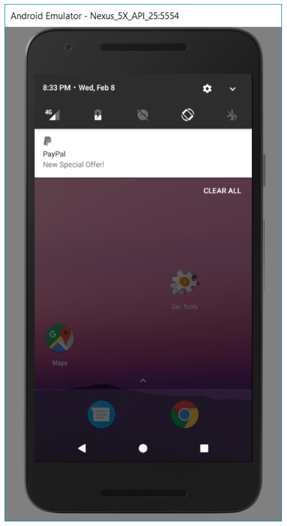
(c) Launching a phone call.


Android Emulator - Nexus_5X_API_25:5554
Search here
Google Maps
To continue, please enter your google password:
OK

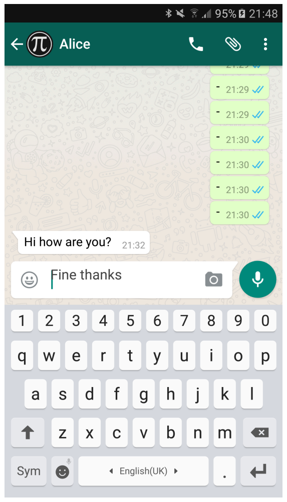
(d) Fake login hovering legitimate apps.



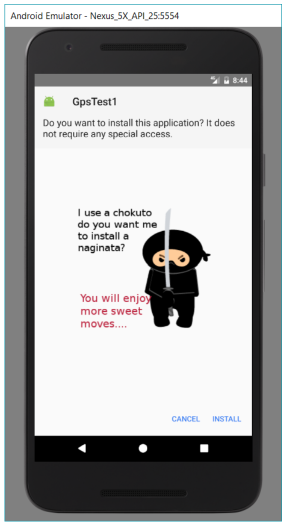
(e) Fake notification and shortcut in the home screen.



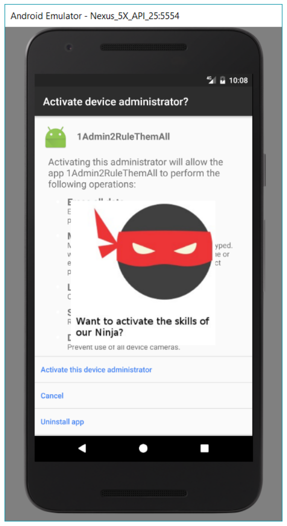
(f) Fake notification as seen in the notification bar.



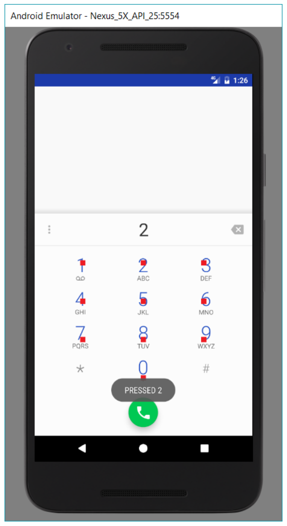
(g) Intercepting user's keyboard input from Whatsapp.


Android Emulator - Nexus_5X_API_25:5554
GpsTest1
Do you want to install this application? It does not require any special access.
I use a chokuto do you want me to install a naginata?
You will enjoy more sweet moves....
CANCEL
INSTALL

(h) Launching package installer.


Android Emulator - Nexus_5X_API_25:5554
Activate device administrator?
1Admin2RuleThemAll
Activating this administrator will allow the app 1Admin2RuleThemAll to perform the following operations:
Want to activate the skills of our Ninja?
Prevent use of all device cameras.
Activate this device administrator
Cancel
Uninstall app

(i) Becoming device administrator.



(j) Revised tapjacking through a grid.

Fig. 2: Various UI redressing attacks.

features described in the previous section, an adversary can easily create a set of activities to cover the screen, leaving a small part of the call button and trick the user in pressing it. A draft example of this approach is illustrated in Figure 2c. Another similar and perhaps more stealth attack would involve sending SMSs to premium numbers.

Sniffing private data from legitimate apps: In this case there are two different attack scenarios. For devices running Android prior to Nougat, an adversary is able to determine which the foreground app is, as presented in the previous section. Should the adversary determine that a specific app would provide him with valuable data *e.g.* credentials, he presents the user with a customised floating activity which covers the legitimate app, requesting private user input. As shown in Figure 2d, the user has no means to determine that the presented activity (shown as a common app dialog) does not belong to the legitimate app. In fact in the illustrated example, Google Maps continues to function in the background as expected, since the floating activity occupies only a specific part of the screen leaving the other parts of the screen unaffected. Considering devices running on Nougat, while the adversary cannot determine which the foreground application is, he can easily trick the user with other methods such as injecting fake notifications or creating fake shortcuts, all mimicking legitimate ones.

Intercepting sensitive input: Should the adversary know which is the foreground application via side channel information, as discussed in the previous section, he can present the user a transparent activity. A typical example is illustrated in Figure 2g where the transparent activity accompanied by a keyboard allows the user to type her message to one of the most widely used messenger applications, Whatsapp. Having intercepted the input, the malicious app displays a message that something went wrong to smoothly return to normal execution.

Fake notifications: Based on the latter restriction in Nougat, about determining the foreground app, we tried a different approach: force the user to open a desired application. To achieve this, we exploited the fake notification mechanism, discussed in the previous section. Therefore, we created a malicious app that downloads dynamically both the notification icons and the notification message. Since the adversary knows which the installed apps are, he can easily create a forged notification for one of the victim's apps. In Figures 2e and 2f we illustrate this in Nougat using PayPal as the target app. As shown in these screenshots, the user has no means to determine that the foreground activity does not belong to PayPal. As already discussed, the notification in Figure 2f may not contain the app name, yet the user most probably will not notice it. Clearly, in Marshmallow, since the name restriction does not apply, the user cannot tell the actual difference, as the forged notification will be identical to the real one. Finally, it should be noted that notifications are used as shortcuts, so the user does not spend much time in trying to determine whether there is a name or not; in the case of Nougat, he will trust the icon.

Fake shortcuts: Another approach to trick the user into launching the forged activity of the malicious app is to create a fake shortcut on the mobile's home screen. While Android has its application menu locked so that applications cannot add more icons, the same does not apply for the home screen. There, any application using the normal permission `INSTALL_SHORTCUT` can create a shortcut with the icon and name of a legitimate and installed application, as described in the previous section. However, the shortcut actually launches the forged activity from the malicious app and not the legitimate one.

Installing applications: Further to performing actions within the scope of the installed applications, an adversary could also trick the user into performing actions within the scope of the operating system per se. For obvious reasons, one would expect that an application would not be allowed to cover activities over them, nonetheless, this is not the case. A profound example is the case of the install manager. Notably, an adversary could download an application from the Internet, by simply using an intent to the browser, or by other means such as utilizing Google Drive, using local files, etc. Practically, using the "Intent" way means that the app does not request Internet permission. Once the download of the APK is finished, the Package Manager is automatically invoked and the malicious app presents the user an activity as in Figure 2h, to trick him and install another app. In the less sinister scenario, the adversary manages to raise his stats, while in the more sinister, the adversary tricks the user into installing an application which has more dangerous payload and the user would have never downloaded from Google Play.

Becoming administrator: In Android 2.2 [3], Google introduced a mechanism that allows users to grant device administration features to specific applications in order to facilitate enterprise applications and to provide them means to apply stricter policies and device management. To this end, an application which is granted this permission can among other features restart the device or wipe its data. To facilitate the installation procedure, Android provides a shortcut so that the application requesting this permission can present the user with this screen. While one would expect that this activity would not be accessible and no one would be able to interact with it once it loses focus, this restriction does not apply. As illustrated in Figure 2i an adversary can cover the activity with the techniques described in the previous section to trick the user into granting some of the most dangerous permissions. Notwithstanding this deceit, the same security gap is present is other highly dangerous activities, *e.g.* installing custom certificates, granting access to user's notifications to name a few. Apparently, the user can be easily tricked into being blocked from his own device, wiping his own data or even giving full remote access to his data.

Tapjacking revisited: The basic concept of most tapjacking attacks in the literature is to create a transparent overlay which exploits a vulnerability in Android's UI to catch the event of user tapping the screen and then passing it to the underlying application. To the best of our knowledge all of these attacks are now obsolete as of Marshmallow. A different approach however is to exploit the grid concept with many sized transparent activities of Figure 1b. The twist in this approach is that we do not try to pass the event to the underlying application, but we exploit the size of users' fingers, as well as the fact that a "sized" activity can even have a surface of a few pixels. Since the activities can also be transparent and can overlay any application, the malicious app can sense where the user's finger is and derive the user's input. Eventually, if the screen is covered by many small transparent activities, touch events will be sensed by the grid, while the interaction with the underlying application will also exist. Notably, in this scenario, the adversary does not need to know the foreground app, as the malicious app logs almost all user tapping so he can later infer sensitive application such as PINs, credentials, keyboard typing etc. To demonstrate the applicability of this attack we created a proof of concept, yet to facilitate the reader, the sized activities are marked red in Figure 2j, but in the original, they are transparent.

5 Conclusions

User interfaces are tightly entwined with user experience, especially regarding user-smartphone interaction. However, the efforts in improving user interfaces may hinder OS security, as app lifecycles are more complex. All the reported attacks, accompanied with the corresponding proof of concept have been already communicated to Google. In some cases, the Android Security team has already responded and provided corresponding software patches, yet other issues are still under investigation.

Considering the notifications and the shortcuts related issues we believe that both users and developers cannot efficiently protect themselves, unless actions in the side of the operating system are taken. Such actions include enforcing the creation of notifications and shortcuts to pass strictly through resource bound parameters. This way, software systems that statically analyse apps installation packages would be able to detect malicious content, such as duplicated third party logos and potentially harmful string values. When Android OS is abused for either tricking users into making *e.g.* unwanted calls, or for escalating malicious apps privileges, we believe that all the involved in these actions activities must be reviewed to handle events when they lose focus and they are overlaid, so that users are notified accordingly. Notably, this mechanism has been partially deployed *e.g.* in Marshmallow's dangerous run-time permissions dialogs. Another alternative would be to disable all OS activity controls when other UI elements appear in front of them. The latter is done in Google Play app, where the presence of a front layer disables some "dangerous" choices, such as the pressing of the "install" app button. The diversity of the two approaches signifies that the problem is known to Google, yet not to its entity, as the patches were applied per case and not generically.

Unfortunately, the aforementioned countermeasures do not apply for the cases of third party apps, therefore the OS could consider adopting them only for the cases where OS activities are involved. That is because many applications provide "floats" in the front most UI screen layer and users find them very usefull, such as the "chatheads" dialogs that are quite common in chatting apps. Implementing the aforementioned solutions could either cause malfunctioning in a large number of applications or continuous annoying alerts, which would negatively affect UX. Subsequently, this raises the need for alternate countermeasures for the third party apps. Towards this direction, a plausible incitement would be to face these security problems differently and enable apps to protect themselves from malicious software. All Android activities are able to "detect" even the slightest changes in their interaction with the user UI, utilizing the Android activities' lifecycle states. That is, overriding the appropriate methods (*e.g.* `onPause` method for detecting dialogs, or `onStop` method when the activities are fully covered by other UI elements) and act accordingly, like disabling or hiding their "sensitive" UI elements, or even alerting their users.

It is also worth noticing that while intents are extremely helpful in providing intercommunication between Android applications, they can be considered a covert channel in terms of permissions, as they provide an out of the loop way of using data and device resources. This is rather important as in static code analysis, one cannot trace this through the corresponding manifest file or the low level API calls, as intents do not map to the framework's methods. This way, they bypass security checks, increasing the complexity to approaches such as Backes et al. [15] as it requires to determine interdependencies between apps.

Concluding, we may state that due to the lack of visible resources that would allow users to determine which the "actual" foremost app is, users have immediate and absolute trust to their OS that the presented apps are the ones they claim to be. It is the authors' strong belief that by providing some more rules and permissions in the Android's UI handling mechanisms in combination with improving security concerns in app development by developers themselves, would lead to the elimination of the majority of the Android UI related security issues raised in this work.

Acknowledgments

This work was supported by the European Commission under the Horizon 2020 Programme (H2020), as part of the *OPERANDO* project (Grant Agreement no. 653704) and is based upon work from COST Action *CRYPTACUS*, supported by COST (European Cooperation in Science and Technology). The authors would like to thank *ElevenPaths* for their valuable feedback and granting them access to Tacyt.

References

1. AlJarrah, A., Shehab, M.: Maintaining user interface integrity on android. In: Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual. vol. 1, pp. 449–458. IEEE (2016)
2. Android Developer: ActivityManager – getRunningTasks. `https://developer.android.com/reference/android/app/ActivityManager.html#getRunningTasks(int)`, date retrieved: 28/03/2017
3. Android Developer: Device administration. `https://developer.android.com/guide/topics/admin/device-admin.html`, date retrieved: 28/03/2017

4. Android Developer: Intent. `https://developer.android.com/reference/android/content/Intent.html`, date retrieved: 28/03/2017
5. Android Developer: Intents and intent filters. `https://developer.android.com/guide/components/intents-filters.html`, date retrieved: 28/03/2017
6. Android Developer: Manifest.permission – READ_EXTERNAL_STORAGE. `https://developer.android.com/reference/android/Manifest.permission.html#READ_EXTERNAL_STORAGE`, date retrieved: 28/03/2017
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 Developer: Multi-window support. `https://developer.android.com/guide/topics/ui/multi-window.html`, date retrieved: 28/03/2017
9. Android Developer: Notification.builder. `https://developer.android.com/reference/android/app/Notification.Builder.html`, date retrieved: 28/03/2017
10. Android Developer: PackageManager – getInstalledApplications. `https://developer.android.com/reference/android/content/pm/PackageManager.html#getInstalledApplications`, date retrieved: 28/03/2017
11. Android Developer: Settings. `https://developer.android.com/reference/android/provider/Settings.html#ACTION_MANAGE_OVERLAY_PERMISSION`, date retrieved: 28/03/2017
12. Android Developer: WindowManager. `https://developer.android.com/reference/android/view/WindowManager.html`, date retrieved: 28/03/2017
13. Aviv, A.J., Gibson, K., Mossop, E., Blaze, M., Smith, J.M.: Smudge attacks on smartphone touch screens. In: Proceedings of the 4th USENIX conference on Offensive technologies. pp. 1–7. USENIX Association (2010)
14. Aviv, A.J., Sapp, B., Blaze, M., Smith, J.M.: Practicality of accelerometer side channels on smartphones. In: Proceedings of the 28th Annual Computer Security Applications Conference. pp. 41–50. ACM (2012)
15. Backes, M., Bugiel, S., Derr, E., McDaniel, P., Octeau, D., Weisgerber, S.: On demystifying the android application framework: Re-visiting android permission specification analysis. In: 25th USENIX Security Symposium (USENIX Security 16). pp. 1101–1118. USENIX Association, Austin, TX (2016)
16. Bianchi, A., Corbetta, J., Invernizzi, L., Fratantonio, Y., Kruegel, C., Vigna, G.: What the app is that? deception and countermeasures in the android user interface. In: Proceedings of the 2015 IEEE Symposium on Security and Privacy. pp. 931–948. IEEE Computer Society (2015)
17. Chen, J., Chen, H., Bauman, E., Lin, Z., Zang, B., Guan, H.: You shouldn't collect my secrets: Thwarting sensitive keystroke leakage in mobile ime apps. In: 24th USENIX Security Symposium (USENIX Security 15). pp. 657–690. USENIX Association, Washington, D.C. (2015)
18. Chen, Q.A., Qian, Z., Mao, Z.M.: Peeking into your app without actually seeing it: Ui state inference and novel android attacks. In: 23rd USENIX Security Symposium (USENIX Security 14). pp. 1037–1052. USENIX Association, San Diego, CA (2014)
19. Faruki, P., Bharmal, A., Laxmi, V., Ganmoor, V., Gaur, M.S., Conti, M., Rajarajan, M.: Android security: a survey of issues, malware penetration, and defenses. IEEE communications surveys & tutorials 17(2), 998–1022
20. Felt, A.P., Finifter, M., Chin, E., Hanna, S., Wagner, D.: A survey of mobile malware in the wild. In: Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices. pp. 3–14. ACM (2011)

21. Fernandes, E., Chen, Q.A., Paupore, J., Essl, G., Halderman, J.A., Mao, Z.M., Prakash, A.: Android ui deception revisited: Attacks and defenses. In: Financial Cryptography and Data Security (2016)
22. Johnson, K.: Revisiting android tapjacking. `https://nvisium.com/blog/2011/05/26/revisiting-android-tapjacking/` (2011)
23. Kartaltepe, E.J., Morales, J.A., Xu, S., Sandhu, R.: Social network-based botnet command-and-control: emerging threats and countermeasures. In: International Conference on Applied Cryptography and Network Security. pp. 511–528. Springer (2010)
24. Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., Mangard, S.: Armageddon: Cache attacks on mobile devices. In: 25th USENIX Security Symposium (USENIX Security 16). pp. 549–564. USENIX Association, Austin, TX (2016)
25. Liu, J., Wang, Y., Kar, G., Chen, Y., Yang, J., Gruteser, M.: Snooping keystrokes with mm-level audio ranging on a single phone. In: Proceedings of the 21st Annual International Conference on Mobile Computing and Networking. pp. 142–154. ACM (2015)
26. Liu, X., Zhou, Z., Diao, W., Li, Z., Zhang, K.: When good becomes evil: Keystroke inference with smartwatch. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. pp. 1273–1285. ACM (2015)
27. Lockheimer, H.: Android and security. `http://googlemobile.blogspot.com/2012/02/android-and-security.html`, date retrieved: 28/03/2017
28. Malisa, L., Kostiainen, K., Och, M., Capkun, S.: Mobile application impersonation detection using dynamic user interface extraction. In: European Symposium on Research in Computer Security. pp. 217–237. Springer (2016)
29. Marforio, C., Masti, R.J., Soriente, C., Kostiainen, K., Capkun, S.: Hardened setup of personalized security indicators to counter phishing attacks in mobile banking. In: Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices. pp. 83–92. ACM (2016)
30. Niemietz, M., Schwenk, J.: UI redressing attacks on Android devices (2012), black-Hat Abu Dhabi
31. Oberheide, J., Miller, C.: Dissecting the android bouncer. In: SummerCon (2012)
32. Richardson, D.: Android tapjacking vulnerability. `https://blog.lookout.com/look-10-007-tapjacking/` (2010)
33. Shukla, D., Kumar, R., Serwadda, A., Phoha, V.V.: Beware, your hands reveal your secrets! In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 904–917. CCS '14, ACM, New York, NY, USA (2014)
34. Simon, L., Anderson, R.: Pin skimmer: Inferring pins through the camera and microphone. In: Proceedings of the Third ACM workshop on Security and privacy in smartphones & mobile devices. pp. 67–78. ACM (2013)
35. Van Bruggen, D.: Studying the impact of security awareness efforts on user behavior. Ph.D. thesis, University Of Notre Dame (2014)
36. Vidas, T., Votipka, D., Christin, N.: All your droid are belong to us: a survey of current android attacks. In: Proceedings of the 5th USENIX conference on Offensive technologies. pp. 10–10. USENIX Association (2011)
37. Wu, L., Brandt, B., Du, X., Ji, B.: Analysis of clickjacking attacks and an effective defense scheme for android devices. In: IEEE Conference on Communications and Network Security. IEEE (2016)
38. Wu, L., Du, X., Wu, J.: Effective defense schemes for phishing attacks on mobile computing platforms. IEEE Transactions on Vehicular Technology 65(8), 6678–6691 (2016)

39. Xu, Z., Bai, K., Zhu, S.: Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors. In: Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks. pp. 113–124. ACM (2012)
40. Ye, G., Tang, Z., Fang, D., Chen, X., Kim, K.I., Taylor, B., Wang, Z.: Cracking android pattern lock in five attempts (2017)
41. Ying, L., Cheng, Y., Lu, Y., Gu, Y., Su, P., Feng, D.: Attacks and defence on android free floating windows. In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security. pp. 759–770. ACM (2016)
42. Zhang, J., Zheng, X., Tang, Z., Xing, T., Chen, X., Fang, D., Li, R., Gong, X., Chen, F.: Privacy leakage in mobile sensing: Your unlock passwords can be leaked through wireless hotspot functionality. Mobile Information Systems 2016 (2016)