How Android's UI Security is Undermined by Accessibility

Anatoli Kalysch
Friedrich-Alexander-Universität
Erlangen-Nürnberg (FAU)
anatoli.kalysch@fau.de

Davide Bove
Friedrich-Alexander-Universität
Erlangen-Nürnberg (FAU)
davide.bove@fau.de

Tilo Müller
Friedrich-Alexander-Universität
Erlangen-Nürnberg (FAU)
tilo.mueller@cs.fau.de

ABSTRACT

Android's accessibility API was designed to assist users with disabilities, or preoccupied users unable to interact with a device, e.g., while driving a car. Nowadays, many Android apps rely on the accessibility API for other purposes, including password managers but also malware. From a security perspective, the accessibility API is precarious as it undermines an otherwise strong principle of sandboxing in Android that separates apps. By means of an accessibility service, apps can interact with the UI elements of another app, including reading from its screen and writing to its text fields. As a consequence, design shortcomings in the accessibility API and other UI features such as overlays have grave security implications.

We reveal flaws in the accessibility design of Android allowing information leakages and denial of service attacks against fully patched systems. With an enabled accessibility service, we are able to sniff sensitive data from apps, including the password of Android's own lock screen. To evaluate the effectiveness of our attacks against third-party apps, we examined the 1100 most downloaded apps from Google Play and found 99.25 % of them to be vulnerable. Although app-level protection measures against these attacks can be implemented, e.g., to prevent information leakage through password fields, the number of affected apps proves that these kind of vulnerabilities must be tackled by Google rather than app developers.

From December 2017 to March 2018, we submitted seven bug reports to Google, from which three have been marked as *won't fix* while four are progressed but ranked with either low severity or no security bulletin class. We conclude our paper with a list of best practices for app-level protections for the time those bugs remain unfixed by Google.

CCS CONCEPTS

• **Security and privacy → Mobile platform security**; **Penetration testing**;

ACM Reference Format:

Anatoli Kalysch, Davide Bove, and Tilo Müller. 2018. How Android's UI Security is Undermined by Accessibility . In *Reversing and Offensive-oriented Trends Symposium (ROOTS '18), November 29–30, 2018, Vienna, Austria.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3289595.3289597

ROOTS '18, November 29–30, 2018, Vienna, Austria

ACM ISBN 978-1-4503-6171-2/18/11…$15.00
https://doi.org/10.1145/3289595.3289597

1 INTRODUCTION

Android's system level security operates upon a sandboxing concept that strongly separates apps. Apps are given an unique user id (UID), thereby shielding both an app's running process as well as its resources from other apps. Android's inter-process communication (IPC) is protected by the kernel driver "Binder", which enforces a permission-based access model. Apps can request particular permissions and only if granted they are allowed access to the respective resources through well defined interfaces [8].

Contrary to that, users can interact with apps seamlessly through a unified UI. And while strong security mechanisms were put into place to ensure the separation of apps and their resources, the effectiveness of those mechanisms fades away when using system services that have nearly the same access rights to the UI as the user—namely *accessibility services (a11y).* Android's accessibility services have access to UI components and can query information about them and perform actions such as tapping the screen and scrolling. As their name implies, those services were originally invented to support users with disabilities. They run as background services remaining completely stealthy while being able to observe and interact with the UI.

Given these characteristics, malware authors soon discovered the potential of accessibility services [5, 12, 15]. *Clickjacking* is often used in conjunction with accessibility services, either to enable an accessibility service, to enable other permissions, or to sniff sensitive information directly, such as passwords. Clickjacking is based on another feature of Android's UI, so-called *overlays*, allowing an app to partly or fully overlap the top activity of an app with its own content [10]. Combining clickjacking attacks with accessibility services, a permission-less app can be used to bootstrap attacks that fully control the UI of a system, by neither requiring privilege escalation attacks, such as root exploits, nor requiring users to consciously approve an a11y service, when running on Android versions prior to 8.

In November 2017, Google planned to reassess all apps using accessibility services in the Play Store. Developers should explain to Google how the use of accessibility services in their apps supports people with disabilities, risking the expulsion from the Play Store [22] if this requirement is not met. The endeavors of Google were cut short, however, due to a public outcry [4] regarding popular apps using accessibility services, such as password managers and anti-theft apps.

Contributions. To investigate the vulnerability of popular Android apps against accessibility services and clickjacking attacks, we conducted an investigation of the 1100 most downloaded apps of the Google Play Store. We found 99.25 % of them to be vulnerable to at least one of our attacks and 90.21% to be vulnerable to all of them.

The high number of affected apps stems from the fact that our attacks exploit design shortcomings of the Android UI that must be tackled by Google rather than by app developers. Nevertheless, we support developers with a list of possible app-level countermeasures because Google is not going to fix all the flaws soon. In detail, our contributions are:

- Vulnerability Discovery: We systematically discover implementation flaws and design shortcomings in Android's UI and a11y system. From December 2017 to March 2018, we submitted seven bug reports to Google. Three were acknowledged but represent issues that will not be fixed, while two are progressed but ranked with low severity, and two are also progressed but ranked with no security bulletin class. So some of the flaws we reveal "work as intended" in the opinion of Google, others will be fixed only slowly. To allow developers to test their apps for those vulnerabilities and consider possible countermeasures, proof of concept (PoC) implementations are published on https://github.com/anatolikalysch/roots_a11y.
- Large-scale Study: To prove the severity of the vulnerabilities, that is their ability to circumvent the principle of sandboxing under Android, we developed PoC attacks and tested them with 1100 of the most popular apps from the Play Store. We tested our PoCs starting with version 6.0 up to version 8.1, including the latest Android versions and security patches available at the time of testing.
- List of Defenses: To counteract UI- and a11y-based attacks, which are already used by malicious apps in the wild [16, 21, 23], we propose defensive techniques for Android app developers. Among others, our techniques allow developers to counteract accessibility event-based sniffing attacks, malicious third-party keyboards and overlay attacks, for the time these flaws remain unfixed by Google.

2 BACKGROUND: EXISTING UI SECURITY MEASURES

In the UI layer of Android, security measures have been implemented to ensure the sandboxing concept remains effective. For example, the Android UI typically prevents apps from accessing the UI elements of other apps without authorization. In the following, we present existing UI security measures that help to separate apps. But with special UI features such as overlays and accessibility services, those security measures fade out.

Window Management. Android's window management builds upon its IPC kernel driver, the Binder. Each app has a unique Binder token which is necessary for window interactions and thereby prevents abuse and information leaks from IPC channels. Only certain system level apps have access to other windows, even when not being the owner of a window, to allow interactions such as copy-and-paste. Android's overlay mechanism, however, allows for non-system windows outside the usual hierarchy, introducing new attack vectors [10]. To reduce the attack surface enabled by overlays, Google introduced a new flag for the *Window* class which allows to dismiss all active non-system overlays. This feature is reserved for system apps and requires special permissions. It is mainly used by confirmation dialogs inside the settings app, e.g., when activating new accessibility services.

UI Permissions. Two permissions are directly related to UI operations, namely the "draw on top" permission which enables an app to create overlays that stay on top of other UI elements, and the "bind accessibility service" permission which enables accessibility services to be started. Both permissions need to be part of an apps Manifest file and be requested once at runtime, with an exception for Google Play Store apps that have the "draw on top" permission granted at installation time. The "draw on top" permission enables apps to create windows that can be designed to pass user interaction to the UI layer below, effectively enabling clickjacking, or to prevent user interaction to reach another UI layer, which was abused by some ransomware variants [2, 24, 26].

The "bind accessibility service" permission allows an app to start its own accessibility service. This operation is security sensitive as accessibility services have access to all active UI elements, even if they do not belong to the app. This includes printed text, window elements, and the ability to interact with UI elements. This permission is given in three steps, contrary to all other permissions. First, the accessibility settings must be opened, which can be done through IPC from inside an app, and then the accessibility service must be selected. Second, the service must be activated through a toggle button. And last, the user is presented with a system alert dialog listing all the capabilities the new accessibility service has.

Secure UI Elements. Preventing screen captures and recordings is another security relevant operation. When displaying sensitive information through the UI, such as credit card numbers, apps can set the `SECURE_FLAG` for a window. This flag prevents screen captures whenever this window is the top activity. On screen recordings, such windows are blacked out on the recording. Also, to hide sensitive information from shoulder surfing, as well as higher privileged services such as accessibility services, password fields are a subclass of the *TextView* class that have a *PasswordTransformationMethod*. The *PasswordTransformationMethod* is responsible for keeping the password hidden as bullet characters and only relaying the hidden password to UI queries.

Input Method Security. Android's *input method framework* (IMF) is responsible for handling the user input. It consists of three components, the *input method manager* (IMM), the *input method editor* (IME), and the client apps requesting the input. While the IME, which is usually a software keyboard, allows users to generate text to be received by the client app, the IMM handles the communication between all components, effectively creating an abstraction layer. The IMM ensures that only one IME and one receiving client app can be active at a time. There are security issues associated with input methods, as they essentially have the ability to monitor what a user enters. Android's design allows for arbitrary third-party keyboards to be used as the default input method editor. To ensure the default keyboard is changed only when deliberately requested by the user, an IME must explicitly be enabled after its installation, and then additionally be set as the default IME via an extra step, similar to the activation of a11y services. Those steps represent UI interactions that cannot be executed programmatically, but by the user only.

3 VULNERABILITIES AND DESIGN SHORTCOMINGS

Unfortunately, the previously described UI security features are not infallible. During an assessment of their effectiveness, we discovered weaknesses and shortcomings in their design, as described in this section.

No Protection from Screen Recordings. As explained above, `FLAG_SECURE` has the ability to prevent screen captures and recordings when being set for the top activity window. Google's official documentation states that activating this flag *treats the content of the window as secure, preventing it from appearing in screen shots or from being viewed on non-secure displays*[1]. Marking a window as secure, however, does not propagate to other interaction interfaces, most prominent example being IMEs. That is, if a window has *EditText* elements where the user types in data such as credentials and credit card information, the activity window itself, including *EditText* fields, is blacked on recordings but the keyboard and the cursor showing which *EditText* is selected are well visible, including highlighted keystrokes. The same is true for pop-up menus such as copy-and-paste and auto completion. While it remains true that the contents of a secured window do not appear on recordings, typed in data can easily be reconstructed from leaking IME information. If attackers have a copy of the targeted app, e.g., installed on their own phones, they can easily correlate input data with the corresponding text fields. Also note that the Android system's own use of the secure flag is very sparse, with even the lock screen being completely visible in screen recordings. Figure 1 demonstrates an example for password input with and without `FLAG_SECURE` being set.

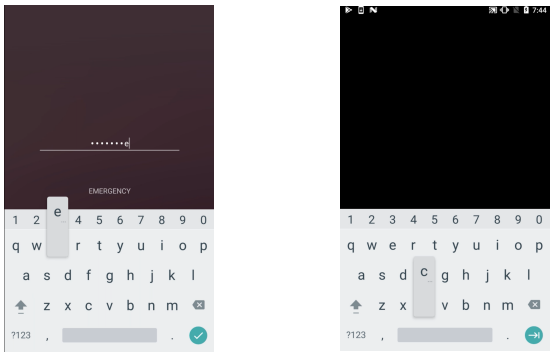

Figure 1: On the left, Android's lock screen without the secure flag is unprotected against screen recordings. On the right, a financial application protects its screen contents with the secure flag, but its IME is unaffected and leaks the password.

Leaking Hidden Passwords. The difference between *EditText* fields containing passwords and other input fields is the presence of a transformation method. The transformation method ensures that the contents of a password field are not propagated to other actors, including shoulder surfers and accessibility services. Instead of the actual characters being printed to the screen or transmitted as accessibility events, bullet characters are visible only. Accessibility events are triggered each time a change in the UI occurs, meaning when selecting a *EditText* field, when a keyboard pops up, and with each character being typed in. Accessibility services can register which events they want to listen to, and the Android system then forwards these types of events. Accessibility events contain all information about the event, e.g., the character that is displayed, and the source of the event, that is the UI element which triggered it. Thanks to the password transformation method, however, sensitive password characters should be represented as bullets.

For usability reasons, Android introduced a feature that shows the last character of a password for 1.5 seconds and then changes it to a bullet. This allows users to control their input while typing in passwords. Entering a character into a password field generates accessibility events that contain the whole text currently displayed inside the field. Not only the UI but also all accessibility events fired from the text field within this 1.5 second time frame contain the last character in plain text. This way Android leaks every password through the accessibility events `TYPE_VIEW_TEXT_SELECTION_CHANGED` and `TYPE_VIEW_TEXT_CHANGED` one character at a time.

Third-Party Keyboard Activation. Google prevents the programmatic activation of newly installed third-party keyboards. After a user installs a new keyboard app, he or she must take further actions to actually use it. First, new keyboards must be activated through the IME settings page. Second, if a new keyboard should become the default IME, this must be configured, too. This way, Google prevents apps from stealthily installing keyboards without explicit authorization by users. This procedure, however, fails to protect users in the light of a11y services able to simulate all user actions. Accessibility services are capable of simulating the complete procedure needed to enable a new keyboard in the IME settings, including the declaration as default input method.

No Synchronization between Settings. Android implements two different ways to change system settings, partly overlapping in functionality but often missing synchronization. The standard way known to most users is via the *Settings* app with sub-menus such as "Apps", "Security" and "Accessibility". Outside of this standard hierarchy, however, Android offers another less known settings page that provides additional information about the phone. These settings are not accessed through conventional means, but by dialing an USSD code (*Unstructured Supplementary Service Data*). For example, dialing the USSD `*#*#4636#*#*` on a Nexus 5X opens a settings menu with the sub-menus "Phone information", "WiFi information" and "Usage statistics". While the menu allows to change critical device settings, such as the preferred network type or disabling the mobile antenna, changes are not reflected in the regular settings menu. In other words, a synchronization between the settings app and these hidden settings is missing. This allows malware to stealthily change settings, e.g., to reroute user traffic or to run denial-of-service attacks. The state of the settings is not reflected to users, even when actively checking the settings menu.

[1] https://developer.android.com/reference/android/view/WindowManager.LayoutParams.html

Overlaying Pictures-in-Pictures (PiP). To mitigate abuses of Android's overlay feature, which allows apps to draw above the top activity, Google introduced a security mechanism that disables overlays when system dialogs are active, the so-called *AppOpsManager*. The *AppOpsManager* takes care that an app cannot overlay permission dialogs with false information looking less suspicious than what is actually requested. This applies to toast messages and system alert windows equally [11]. While the *AppsOpsManager* is commendable from a security perspective, it apparently lacks ongoing development when new Android features are introduced. The introduction of Android 8 welcomed a new UI feature called picture-in-picture mode (PiP), which is also able to overlay the top activity, but is not blocked by the *AppsOpsManager* when displayed over system dialogs. PiP allows a video currently being shown in one window to be displayed on top of the whole UI when being minimized. The effect is similar to overlays, however, PiPs are not realized with any of the previously known overlay windows, thereby mitigating the *AppOpsManager* API. See figure 2 for a screenshot.

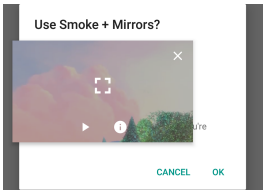

Figure 2: The vulnerability of the *AppOpsManager* API against overlaying picture-in-picture windows (PiP).

Misleading A11y Capabilities. To activate an a11y service, users need to explicitly select it in the settings menu and confirm the activation with a pop-up dialog. The dialog displays all capabilities the new a11y service has and should allow users to make an informed decision about the activation. See Figure 3 for a screenshot. Note that each listed capability directly corresponds to a configuration item of the a11y service. Requesting certain a11y capabilities results in additional items being displayed in the pop-up dialog the user needs to confirm. We argue that the description the user sees is not representative for the actual power that an a11y service has. To cut down the description of capabilities an a11y service gains to just one sentence is not a trivial task, but in its current form, the information given is misleading. For example, the flag *canRequestFilterKeyEvents* leads to the sentence "Observe text you type: Includes personal data such as credit card numbers and passwords". This capability allows an a11y service to receive events such as key strokes from a keyboard. However, a11y services that do not set *canRequestFilterKeyEvents*, and instead listen to all available accessibility events by setting *accessibilityEventTypes="typeAllMask"* have effectively the same power without generating this scary sentence.

4 ATTACKS

In the following, we describe three proof-of-concept attacks we implemented for our study aiming at sensitive input data such as credentials.

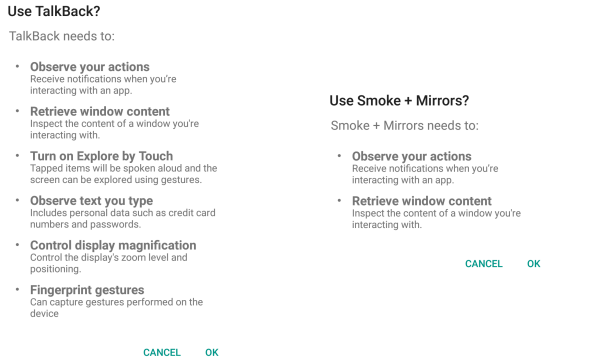

Figure 3: Activation of Google's TalkBack (on the left) and our own a11y service (on the right). While TalkBack actively admits that it may sniff passwords our implementation does not, despite having even more capabilities.

Threat Model. We assume an uncompromised Android device not being subject to privilege escalation attacks such as rooting. We also assume that the owner of a device installed a malicious app that runs inside the Android sandbox like any other app, without special privileges. It has been shown by prior work that malicious apps can enter third-party app stores and even the Play Store by mitigating its defense measures [1, 10, 14, 19], and third-party markets are another source of malware. Furthermore, some of our attacks require that we activate an app's a11y service, which can be done without a user's conscious approval for Android versions prior to 8 as has been shown by Fratantonio et al.. Starting with Android 8, however, this attack sequence is thwarted by Google, as the *AppOpsManager* API deactivates all overlays during the last step in the activation sequence.

This leaves social engineering as the only possibility to enable a11y services on Android 8, according to our knowledge. Kraunelis et al. and Fernandes et al. covered social engineering-based approaches to enable a11y services that still work on current Android versions, as the activation procedure has not changed [9, 17]. In addition, malware is increasingly becoming aware of the a11y API and relying on social engineering approaches, e.g. in the case of a trojaned Google Play app that needed the a11y permission to "function properly" [23]. Those attacks could be further enhanced by misleading system messages, as explained in section 3. Since users are warned about the capabilities of an a11y service, but such warnings do not reflect the actual capabilities, users could be convinced to accept new services more easily. The comparison of Google's TalkBack with our own a11y service, depicted in Figure 3, is a good example as it paints our implementation as less capable than Google's service, while the opposite is actually true.

Some of the attacks presented below require a certain setup procedure, such as confirming a screen recording dialog or installing a new default IME. Those procedures are carried out by an a11y service once it has been enabled. But while some procedures are minimal and can be handled via a11y services within milliseconds, such as confirming a pop-up dialog, others are too complex and will be recognized by users. In all cases, however, we can use screen-wide overlays to hide the malicious behavior behind a facade.

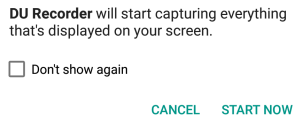

Figure 4: The pop-up window serving as the security measure against unwanted screen recordings which generates a11y events and is therefore usable by a11y services.

A11y Event Sniffing. An a11y service needs to declare all events it wants to listen to. Usually an a11y service has a specific use case in mind that produces a set of accessibility events such as `TYPE_WINDOW_STATE_CHANGED`, to react when a window changes its appearance. Alternatively, an a11y service can simply listen to all accessibility events by setting the flag *accessibilityEventTypes=typeAllMask*. A user is unable to query which events an a11y service is listening to and listening to all accessibility events does not generate an additional capability in the a11y service pop-up dialog, shown in Figure 3. Accessibility events carry the source of the event, an *AccessibilityNodeInfo* object. Using this object as a starting point, we can traverse the currently active UI elements and, by exploiting design issues in the password transformation method as explained in section 3, we can create an input sniffer for all user input that categorizes data into username/password pairs and payment infos.

An attacker now needs to create an a11y service listening to all or at least text field related a11y events[2]. Due to the 'leaking hidden passwords' vulnerability introduced in section 3 even passwords will generate an a11y event for each inputted character containing the last character in plain text, allowing to sniff the whole password. Other text fields are not protected in any way and reveal the whole content through each ally event. The a11y service should also allow for a form of information exfiltration, e.g., to an attacker controlled server, irc, or even pastebin[3].

With the a11y service prepared in such a way an attacker would need the user to activate the service, e.g., by repackaging a popular app with his a11y service and requiring the activation to run the app. Now the a11y service can sniff any events, even distinguish password fields by calling the *isPassword()* method on the incoming *AccessibilityNodeInfo* objects. The credentials or even the whole log can then be exfiltrated through the previously chosen channel.

Screen Recording. Android's security mechanism for screen recordings is based on a pop-up dialog, asking the user to confirm that his or her screen will be recorded. An exemplary dialog is presented in figure 4. Contrary to other system resources, no additional permission must be requested in the Android Manifest file. Not only can an a11y service confirm the dialog, and thereby start the screen recording, but also can the mark allowing future requests be checked easily. After confirmation, the a11y service gains the ability to start and stop a screen recording at any time. In our PoC attack, we use the package names inside accessibility events as an indicator, e.g., when PayPal's package name appears in accessibility event, we start the recording and stop it as soon as the top activity app changes. This allows us to record the whole interaction a user has with specific apps, and upload the videos to an attacker-controlled system once connected to a WiFi network.

Malicious Third-Party IME. Google's own IME was recently changed to prevent sending the current input character via accessibility events. However, users can still change their default IME, meaning aside from Google's keyboard another one can be activated. For our attack, we created an IME similar to the UI of Google's keyboard, and introduced additional keystroke sniffing. The keyboard is packaged alongside our a11y service and after the a11y service gets enabled, we activate the keyboard and install it as default IME programmatically, allowing to sniff any input a user generates, including passwords, financial info and more.

5 VULNERABILITY ASSESSMENT

To assess how many apps are affected by the UI design issues and attacks described above, we decided to test the most downloaded apps from the Google Play Store. Specifically we wanted to test if and how the most typical apps are affected regarding sniffing attacks against login credentials.

5.1 Test Setup

For our assessment, we focused on Android versions 6.0 to 8.1. As of mid 2018, the distribution of Android versions is 23.5% for Android 6, 30.8% for Android 7 and 12.1% for Android 8[4]. Remaining devices use even older Android versions. To create a significantly sized dataset, we implemented a Play Store crawler and downloaded the first 500 apps from the "top charts" category. Additionally, we focused on the categories *business, communication, dating, entertainment, finance, games, health, shopping, social and travel*, and also downloaded popular apps from these categories if they were not processed yet, yielding a total of 1100 apps. The test device was a Nexus 5X with 32 GB of storage. Furthermore, we used crawlers for the third-party app stores *f-droid.org, uptodown.com, shouji.baidu.com, eoemarket.com,* and *apkpure.com* to create datasets of the 500 most downloaded APKs for *each* 3rd party store.

We focus on login screens of apps only since to bypass a login screen, e.g., to enter other information such as credit card numbers, a valid account per app would be required. Opening up a new account per app can not be automated well, sometimes requiring days until valid credentials are obtained. For these reasons we manually inputted fake credentials into the apps while our PoC attack implementations were sniffing credentials in the background. Afterwards we compared the sniffed credentials with the fake credentials to assess the effectiveness of the data leak.

In an initial run of our dataset, we determined that 722 apps have a login field. Furthermore, 81 apps did not have their own login screen but supported logins via third-party accounts. Testing malicious keyboards, screen recordings and accessibility events simultaneously is possible as they do not overlap as attack vectors. If an app came packaged with its own keyboard, or even enforced an exclusive IME for an input field, we rated it as immune to the

[2] `TYPE_VIEW_TEXT_CHANGED` and `TYPE_VIEW_TEXT_SELECTION_CHANGED`.
[3] https://pastebin.com/

[4] According to Google's distribution dashboard on https://developer.android.com/about/dashboards/.

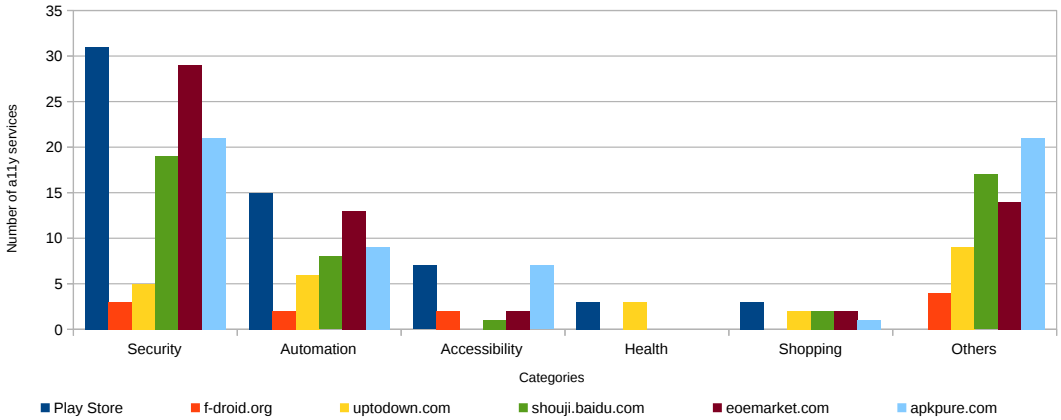

Figure 5: Number of a11y service per app category. This graphic shows that implementing an a11y service is not uncommon for legitimate apps. The sample size for the Play Store was 1100 and the third-party stores was 500 apps each.

keyboard attack. Likewise, if an app filtered or disabled outgoing accessibility events, we rated it as immune against data leakages via accessibility events. If `FLAG_SECURE` was set for the login window, we still checked whether the default keyboard appears to be visible on screen recordings.

5.2 Use of A11y Services

We briefly clarify how and in which apps a11y services are used today. Overall we encountered 59 a11y services in the Play Store dataset (1100 apps), 11 on f-droid, 25 on uptodown, 47 on shouji.baidu, 60 on eoemarket, and 59 on apkpure. As listed in Fig. 5, we grouped the a11y services into the categories "security", "automation", "accessibility", "health and fitness", "shopping", and "others". "Security" contains all apps providing device security, like anti-virus apps, file cleaners and password managers. "Automation" includes all apps that use triggers to automate everyday tasks, like location or time dependent UI automation. The categories "shopping" and "health and fitness" contain apps that allow the user to buy articles from online shops or have a health focus, respectively. The Play Store already categorizes its apps according to these categories. For third-party store apps not present in the Play Store, we performed a manual categorization. There were some apps with Chinese titles and UI which we could not associate correctly, and choose to distribute them into the "other" category.

Fig. 5 displays the categories and the number of a11y services in each category for each app store. The most prominent category for a11y services is security, closely followed by automation. Quite striking is that nearly all anti-virus apps had their own a11y service, often to support protecting web sessions and file cleaning. To sum up the use of a11y services in today's apps, improving the usability for disabled users is rarely the intended use case.

5.3 Test Results

We analyzed the whole dataset from Google Play (1100 apps) with respect to its vulnerability against the attack types described in section 4. Testing the 1100 apps provided a good trade-off, as it allowed us to make statistically significant statements while keeping the manual overhead manageable, as credentials needed to be manually entered for each app. We discovered that 38 apps deactivated accessibility events while further 21 filtered the events in a way that makes extracting the password impossible. Both methods prevent a11y event sniffing and pose an effective countermeasure against it. On Android 7.0 and 7.1.2, WebViews pose another limitation for event sniffing because the password is fully transformed, including the last character. In our dataset, 144 apps used a WebView to authenticate the user, hence making them immune to event sniffing. This behavior was removed with Android 8.0 and 8.1, where WebViews fully leak passwords again.

A possible countermeasure against the screen recording attack is `FLAG_SECURE`. We encountered 72 apps that used this flag to secure their screen's content. However, as explained above, `FLAG_SECURE` does not protect the keyboard, thus allowing attackers to extract typed in credentials despite this flag being set. The cursor is also not protected by `FLAG_SECURE`, thus leaking information about which field is currently selected. An effective countermeasure to prevent these kind of attacks is to create an in-app keyboard, at least for password fields and logins. We encountered 10 apps with their own keyboard, with one creating an all-purpose IME and asking the user to use it, while the rest used exclusive in-apps IMEs. The implementation of these IMEs, however, was often flawed and sometimes worse than Google's default IME. They leaked the entered characters via a11y events and `FLAG_SECURE` was not set in 8 out of 10 cases. Only 2 in-app IMEs proved to be invulnerable to both a11y-based sniffing and screen recordings. We also encountered two apps warning the user with a pop-up dialog about active a11y service, typically referred to as "screen readers". This seems quite ineffective, as an a11y service can interact with the pop-up and confirm it.

Table 1 summarizes the state of vulnerable login screens per app category. While app-based countermeasures as a quick-fix exist, those are not wide-spread. We found 99.25 % of all app logins to be vulnerable to at least one of our attacks, and 90.21% to be vulnerable to all of them. Above that, all third-party login screens, including

Category	Number of apps with a login	Percentage of logins vulnerable against		
		A11y Events	*Screen Records*	*Malicious IMEs*
Business	116	100%	100%	100%
Communication	47	100%	100%	100%
Dating	63	100%	100%	100%
Entertainment	58	100%	100%	100%
Finance	172	84.9%	96.5%	94.2%
Games	104	95.2%	100%	100%
Health	57	98.3%	100%	100%
Shopping	42	95.2%	100%	100%
Social	99	100%	100%	100%
Travel	45	97.7%	100%	100%
Summary	803	95.6%	99.3%	98.8%

Table 1: Out of 1100 apps 803 had a login screen, most of them being vulnerable. This table divides the detected login screens into their subsequent category and provides an overview of what percentage of these logins screens were vulnerable against data leakages by a11y event sniffing, screen recordings, and malicious IMEs.

Facebook and Google logins, were found vulnerable against all attacks, with Facebook being employed most often. Fig. 6 summarizes which and how many countermeasures had been deployed per app category. Note that when app-level countermeasures have been employed, the app likely belongs to the category "Finance". Above that, for mobile banking apps, two-factor authentication (2FA) is often used, thus sniffing credentials alone often being not sufficient. If the second factor is a second app running on the same device, both apps must be protected against a11y attacks. If SMS is used as a second factor, the protection is reduced considerably because SMS are readable through UI interaction by a11y services using Android's default SMS app.

5.4 Responsible Disclosure

Naturally Google must be the party to fix the UI bugs disclosed above, not the app developers. We submitted bug reports to Google starting in December 2017. The reports include vulnerabilities and attacks concerning the broken password transformation method (PTM), screen recordings (SR), and malicious IME activation (mIME), which were reported on December 1st, 14th and 18th, respectively. The PTM and the mIME were assigned within a week and have remained in this status since then. Both security reports were assigned a low "Android Security Reward" (ASR) severity, and remain unfixed at the time of writing. SR of IMEs were set to "Won't Fix" after initial review and given an ASR severity of "Non-Security Bulletin Class". "Non-Security Bulletin Class" is assigned to all bugs that will not be featured on Google's security bulletins.

Four additional bug reports covering the insufficient setting synchronization, PiP overlays, misleading a11y capabilities, and a11y-based DoS attacks were reported in February/March 2018. While the DoS attacks and misleading a11y capabilities were assigned a "Won't Fix", the other two reports were assigned but have not been updated since. Note that the DoS attacks we reported to Google have been omitted due to spacial constrains, but relies on the missing synchronization between the settings vulnerability described in section 3. Our line of argument was that, as described in Sect. 3, hidden settings allow the configuration of the mobile network and WiFi settings. Exploiting missing synchronizations

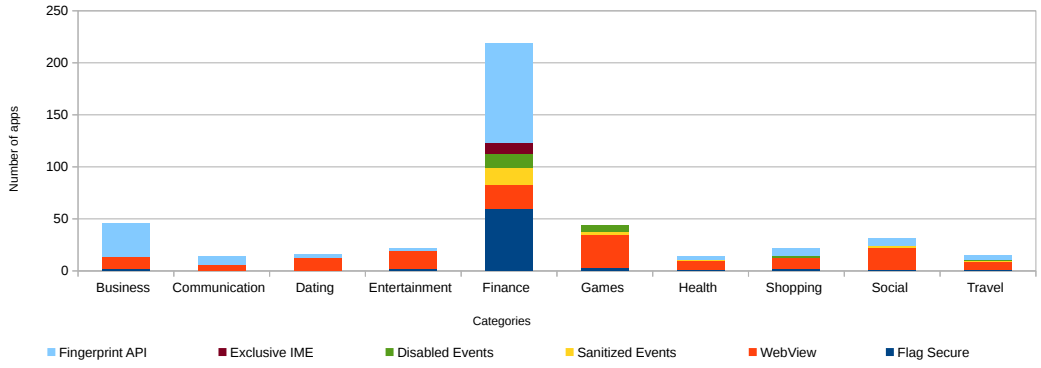

Figure 6: Employed security mecanisms per category. Applying only one of these protective mechanism is usually not enough to protect against all our attacks.

with the regular settings, we created DoS attacks on the phone's ability to interact with the mobile network by disabling a device's mobile antenna or changing the default SMS center, for example.

From a security point of view this puts application developers in charge of securing their own applications. Considering we found all tested Android versions to be vulnerable and 99.25 % of the apps in our dataset to be vulnerable to attacks this shows a great potential for improvements. Most developers might simply not be aware of a11y based attacks or trusting the Android OS to secure them from these attack vectors. The past has seen exploitation of this API in academia ([10, 17]) and malware research alike ([21, 23]) so the need for a clear baseline of who is responsible for a11y security is apparent but can only be established in dialog with AOSP maintainers.

6 COUNTERMEASURES

In the following, we propose countermeasures for app developers who want to protect against the attacks shown in section 4 independent of Google. With the proposal of in-app protection measures, we try to deal with the situation that Google will not fix some of the bugs we reported. We focus on countermeasures that can be implemented by unprivileged apps, not requiring system privileges or root. We additionally evaluate relevant attacks from previous work for their malicious capabilities up to Android 8.1 to show which attacks endanger third party apps on the most recent Android version and their countermeasures.

A11y Event Filtering. Since any default view element generates events that can be sniffed by an a11y service, our first countermeasure aims at filtering these events explicitly. To prevent a11y events either custom views can be employed or event propagation can be disabled. A *View* object implements the *AccessibilityEventSource* interface, which is responsible for sending events to an a11y service. By overwriting the methods *sendAccessibilityEvent* and *sendAccessibilityEventUnchecked* it is possible to remove the default propagation for custom views which inherit from *View*. For already existing views, however, such as *LinearLayout* and *RelativeLayout*, modifying the source code is not possible. Thus, developers can use the *View.AccessibilityDelegate* class to modify the accessibility behavior of the view. When set, it is used to delegate calls to the *sendAccessibilityEvent** methods to an object the developer controls, effectively allowing the event generation to be prevented without modifying the source.

Behavior Listeners and A11y Services. While a11y services pose a powerful tool and are quite adept at simulating user behavior, i.e., taking UI actions for the user we found ways to distinguish and prevent a11y services from taking UI actions if required. Specifically, a11y services have two major limitations when the `ACTION_CLICK` is performed on an UI element. First, no click coordinates are available from this event contrary to a user generated click that contains the X and Y coordinates. Checking for the click coordinates can very well determine if a real user was the initiator. Second, an a11y service can only trigger *OnClickListeners* and is unable to trigger *OnTouchListener*. This results in app logic encoded in, e.g., *TouchDown* or *TouchUp MotionEvents* from never being triggered by a11y services at all.

In-App Keyboards. A dedicated keyboard per app, implementing particular security measures, can counter data leaks through screen recordings and malicious third-party keyboards alike. While a general IME, which the user installs as a separate app, might be a viable solution to prevent screen recordings, it depends on the user to take security actions. An app-exclusive IME bound to specific input fields, e.g., the password input, can provide protection independently of the user behavior. Such keyboard must first be secured with the `FLAG_SECURE` and second employ event filtering, as explained above. Some apps already have a custom app keyboard implemented, however, during our vulnerability assessment, we did not encounter any custom IME that had its a11y events sanitized, and less than half had the secure flag set.

Window Punching. Recent advances in Android UI-based attacks combined a11y services with overlays, by either overlaying the whole app [9] or its login fields [10]. While deactivating a11y events would ensure that the user detects the attack, the detection would happen only after entering credentials when the information leak already happened. A proactive way to prevent users accidentally entering input into overlay screen builds on the "window punching" technique proposed by AlJarrah and Shehab [3]. Using the *Instrumentation* library of the Android SDK to simulate touch events in random intervals, it can be tested whether input fields are overlayed. Whenever a touch event is fired and hits a window not owned by the current app, a *SecurityException* is thrown by the app.

Fingerprint Authentication. Constitutes an alternative authentication mechanisms, not allowing for credential sniffing, and offering high usability. The use of Android's fingerprint API protects against many a11y-based attacks. There were, however, no apps in our dataset offering only a fingerprint-based authentication. Naturally an appropriate device with fingerprint sensor and hardware-backed key storage is needed, as the API is used in conjunction with the *Android Keystore System* to store cryptographic keys on the device.

Android 8 UI Security Features. Android 8 introduced additional countermeasures against UI-based attacks. The impact of these protection mechanisms has not been assessed so far and it is thus unknown if the previously proposed a11y-based attacks still pose a threat to the latest Android version. To fill this gap we evaluated our proposed attacks and re-implementations of previously proposed a11y and overlay-based attacks for the Android versions 6.0 to 8.1. We focus on selected attacks that target third party applications, e.g., overlays over a login screen or sniffing credentials, and not attacks on the Android OS itself like the installation of an app and enabling its permissions with the help of overlays and a11y services.

Table 2 provides an overview of our investigation. A11y focused attacks seem not affected by the most recent version of Android, resulting in an incredibly powerful tool for attackers. However, apps still have possibilities to counteract this attack vector on their own. A11y event sanitizing offers powerful capabilities because it allows to define custom behavior for events, e.g., clean a11y events of any relevant information, which is helpful for credential fields, or even suppress a11y events which can prevent a11y-based ad hijacking. To counteract the overlay and a11y assisted password stealing introduced by Fratantonio et al. window punching can

Attack	Vulnerable Android Versions					Possible Countermeasures
	6.0	7.0	7.1.2	8.0	8.1	
A11y Event Sniffing	✗	✓	✓	✓	✓	a11y event sanitizing, fingerprint auth.
A11y Screen Recording	✓	✓	✓	✓	✓	secure flag *and* in-app keyboard
A11y-enabled Malicious IME	✓	✓	✓	✓	✓	in-app keyboard *and* behavior listeners
A11y-based Ad Hijacking [10]	✓	✓	✓	✓	✓	a11y event sanitizing
Overlay and a11y assisted password stealing [10]	✓	✓	✓	✓	✓	a11y event sanitizing, window punching
Keyboard App Hijacking [10]	(✓)	(✓)	(✓)	✗	✗	in-app keyboard *or* enforcing Gboard update
Full App Passthrough / Clickable Overlays [17]	✓	✓	✓	✓*	✓*	window punching
Partial App Clickable Overlays [9]	✓	✓	✓	✓*	✓*	window punching
Context-aware Clickjacking / Hiding [10]	✓	✓	✓	✓*	✓*	window punching
Keystroke Inference [10]	✓	✓	✓	✗	✗	in-app keyboard *and* window punching

Table 2: A11y and overlay-based attacks presented in this paper and in previous work. A check mark ✓ signals a vulnerable Android OS version while the x mark ✗ signals effective protection mechanisms by the OS. In addition to our own attacks we reimplemented and evaluated attacks from prior work for Android 8 and 8.1 which introduced new UI security mechanisms.

be used against the overlays and a11y event sanitizing can prevent the events and their *AccessibilityNodeInfo* objects of the login activity and its UI elements to be leaked to the malicious a11y service. Without them an a11y service is unable to act upon UI elements effectively preventing this attack. Using behavior listeners allows to ensure certain UI elements can not be triggered by an a11y service, e.g., in the case of requiring user consent a button could implement an *OnTouchListener* and only act in case of a *TouchUp MotionEvent*. The downside quite obviously is the increased engineering effort, as several use cases need to be analyzed and attack vectors guarded against. Depending on the number of UI elements involved in critical app operations this introduces a considerable overhead. The Keyboard App Hijacking [10] which used the a11y flag `FLAG_RETRIEVE_INTERACTIVE_WINDOWS` was fixed for Android's stock input method, the Gboard app. This also results in previous versions being secured as well, as long as they have updated the application.

Overlay-based attacks see themselves confronted with new security mechanisms. Passthrough, clickable, and "hole" overlay-based attacks will result in a notification in Android's notification bar. The overlay will still be displayed however, so the effectiveness of this countermeasure hinges on whether the user is able to notice the new notification or not. Additionally, Android 8 prevents overlays altogether during certain critical system operations, e.g., enabling an a11y service. However, third party applications currently have no possibility to disable overlays during their critical operations. Lastly, the side-channel Fratantonio et al. used to infer information for their invisible grid has been fixed, preventing the keystroke inference attack on Android 8. Window punching can be very well used to counteract overlays. The punches need to be quite random in terms of timing intervals and placement inside the UI to prevent timed overlay attacks.

7 RELATED WORK

Lin et al. [18] realized UI-based credential leakages by utilizing Android's Debug Bridge (ADB). They present a PoC implementation that is able to derive the correct timing to sniff user credentials through screen shots initiated through ADB [18]. Conceptually, this is closest to our a11y screen recording, however, we leverage a11y services and the MediaProjection API to initiate screen recordings due to the inability to use screen shots during `FLAG_SECURE` enabled windows.

Kraunelis et al. [17] presented malware using an a11y service to enhance the effectiveness of phishing attacks. Similar to our work, the introduced attack leveraged the a11y API to sniff user credentials. Contrary to our work, however, Kraunelis et al. use a11y services as a side channel to detect recently launched apps and present a fake login activity to the user. The sniffing operation takes place in the fake login activity which needs to be adapted for each app, while our approach is independent from the target app and relies on the a11y API directly for its sniffing purposes.

Jang et al. [13] analyzed accessibility features of Microsoft Windows, Ubuntu Linux, iOS and Android. Their evaluation led to the discovery of three design flaws in Android's a11y API. We focus on Android versions 6.0 – 8.1 while Jang et al. [13] focused on 4.4 which was prior to major design changes, e.g., the introduction of the Android RunTime and runtime permissions. Also, we complete our assessment with a study on the vulnerability of Google Play Store apps to our attacks, possible defenses, and the use of a11y services in the wild.

Fratantonio et al. [10] presented a security assessment of Android's UI and uncovered design flaws and several innovative attacks which combine the use of UI-elements and a11y services. In addition, they describe overlay-based attacks that can be used to bootstrap the activation of an a11y service. Contrary to Fratantonio et al., our attacks rely less on overlays and focus more on a11y services. E.g., while the password stealing described in this document listens to a11y events, their password stealing leverages overlay elements over the input fields. Resulting from our focus on a11y services, we conduct an in-depth investigation on the use of the a11y API including Google's Play Store and third party app stores. So we focus on the current state of the Google Play Store and assess the effect to the most downloaded apps while Fratantonio et al. focus on the effectiveness of their attacks in a user study.

Clickjacking attacks were prominently featured in research and malware reports utilizing Android's overlay feature to obscure underlying windows. In 2012, Niemietz and Schwenk [20] were the first to identify UI-based clickjacking attacks on Android devices. By adopting known web-based clickjacking techniques, they examined which attacks can be ported to mobile devices. While the focus of their work were mobile web browsers, they also investigated an app-based approach, using full-screen toast overlays to trick users into dialing a premium phone number. Further research that aims at tricking users into giving away personal information, or granting permissions to malicious apps, by UI attacks can be categorized in overlay-based attacks [2, 24, 25] and the use of side channels [6, 7, 9] for improved phishing attacks. While clickjacking attacks may allow credential sniffing, they need to be tailored to a specific app. This is not the case when exploiting the a11y API instead.

8 CONCLUSION

To summarize, we presented vulnerabilities and design shortcomings in Android's UI design and the accessibility API in particular. Exploiting the vulnerabilities, we created proof-of-concept attacks that allow an attacker to steal sensitive information of third-party apps such as login credentials. We have proven the practicability of our approach on Android 6.0 to 8.1 by testing a dataset of 1100 apps from the Play Store. We found all tested Android versions to be vulnerable and 99.25 % of the apps in the dataset to be affected. To facilitate bug fixing of the discovered vulnerabilities, we reported them to Google resulting in seven reports from December 2017 to March 2018. To help the developers of apps with protecting their apps, despite Google's reaction, we presented countermeasures that allow mitigating these attacks and attacks from previous work on vulnerable Android versions.

To conclude, Android suffers from a series of UI designs that have deliberately been introduced by Google but pose a security threat beyond their intended use case. This applies to both the feature of overlays, including PiP, as well as the omnipotent accessibility API. From a usability point of view, one could argue these "features" increase the user experience, but Apple iOS, typically not perceived less usable by end users, goes without them. Furthermore, Apple iOS enforces the system's default keyboard for all password fields – a reasonable protection missing on Android.

REFERENCES

[1] Vitor Afonso, Anatoli Kalysch, Tilo Müller, Daniela Oliveira, André Grégio, and Paulo Lício de Geus. 2018. Lumus: Dynamically Uncovering Evasive Android Applications. In *International Conference on Information Security*. Springer, 47–66.

[2] Efthimios Alepis and Constantinos Patsakis. 2017. Trapped by the UI: The Android case. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 334–354.

[3] Abeer AlJarrah and Mohamed Shehab. 2016. Maintaining user interface integrity on Android. In *Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual*, Vol. 1. IEEE, 449–458.

[4] Ron Amadeo. 2017. Public outcry causes Google to rethink banning powerful "accessibility" apps. (2017). https://arstechnica.com/?post_type=post&p=1229629, accessed on 27. July 2018.

[5] Yair Amit. 2016. Accessibility Clickjacking – Android Malware Evolution. (2016). https://www.symantec.com/connect/blogs/accessibility-clickjacking-android-malware-evolution, accessed on 11. August 2018.

[6] Antonio Bianchi, Jacopo Corbetta, Luca Invernizzi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. 2015. What the app is that? Deception and countermeasures in the Android user interface. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 931–948.

[7] Qi Alfred Chen, Zhiyun Qian, and Zhuoqing Morley Mao. 2014. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks.. In *USENIX Security Symposium*. 1037–1052.

[8] Nikolay Elenkov. 2014. *Android security internals: An in-depth guide to Android's security architecture*. No Starch Press.

[9] Earlence Fernandes, Qi Alfred Chen, Justin Paupore, Georg Essl, J Alex Halderman, Z Morley Mao, and Atul Prakash. 2016. Android UI deception revisited: Attacks and defenses. In *International Conference on Financial Cryptography and Data Security*. Springer, 41–59.

[10] Yanick Fratantonio, Chenxiong Qian, Simon P Chung, and Wenke Lee. 2017. Cloak and Dagger: from two permissions to complete control of the UI feedback loop. In *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 1041–1057.

[11] Svet Ganov. 2016. https://android.googlesource.com/platform/packages/apps/PackageInstaller/+/00b8a2. (Feb 2016).

[12] Dan Goodin. 2018. New Android Malware with never before seen spying capabilities. (2018). https://arstechnica.com/?post_type=post&p=1244897, accessed on 17. August 2018.

[13] Yeongjin Jang, Chengyu Song, Simon P Chung, Tielei Wang, and Wenke Lee. 2014. A11y attacks: Exploiting accessibility in operating systems. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 103–115.

[14] Anatoli Kalysch, Oskar Milisterfer, Mykolai Protsenko, and Tilo Müller. 2018. Tackling Androids Native Library Malware with Robust, Efficient and Accurate Similarity Measures. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*. ACM, 58.

[15] Swati Khandelwal. 2017. Ransomware Not Just Encrypts Your Android But Also Changes PIN Lock. (2017). https://thehackernews.com/2017/10/android-ransomware-pin.html, accessed on 20. August 2018.

[16] Eduard Kovacs. 2017. Google Play Malware abuses Accessibility Services. (2017). https://www.securityweek.com/android-malware-found-google-play-abuses-accessibility-service, accessed on 24. August 2018.

[17] Joshua Kraunelis, Yinjie Chen, Zhen Ling, Xinwen Fu, and Wei Zhao. 2013. On malware leveraging the Android accessibility framework. In *International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*. Springer, 512–523.

[18] Chia-Chi Lin, Hongyang Li, Xiao-yong Zhou, and XiaoFeng Wang. 2014. Screenmilker: How to Milk Your Android Screen for Secrets.

[19] Dominik Maier, Tilo Müller, and Mykolai Protsenko. 2014. Divide-and-Conquer: Why Android Malware cannot be stopped. In *Proceedings of the 9th International Conference on Availability, Reliability and Security (ARES)*. SBA Research, Fribourg, Switzerland.

[20] Marcus Niemietz and Jörg Schwenk. 2012. Ui redressing attacks on android devices. *Black Hat Abu Dhabi* (2012).

[21] Danny Palmer. 2017. Android banking malware steals data by exploiting smartphone accessibility services. (2017). https://www.zdnet.com/article/this-android-banking-malware-steals-data-by-exploiting-smartphone-accessibility-services/, accessed on 15. August 2018.

[22] Mishaal Rahman. 2017. Google is Threatening to Remove Apps with Accessibility Services from the Play Store. (2017). https://www.xda-developers.com/google-threatening-removal-accessibility-services-play-store/, accessed on 17. August 2018.

[23] Gaurav Shinde. 2017. Malware on Google Play abusing Accessibility Service. (2017). https://www.zscaler.com/blogs/research/malware-google-play-abusing-accessibility-service, accessed on 14. June 2018.

[24] Longfei Wu, Benjamin Brandt, Xiaojiang Du, and Bo Ji. 2016. Analysis of clickjacking attacks and an effective defense scheme for Android devices. In *Communications and Network Security (CNS), 2016 IEEE Conference on*. IEEE, 55–63.

[25] Lingyun Ying, Yao Cheng, Yemian Lu, Yacong Gu, Purui Su, and Dengguo Feng. 2016. Attacks and defence on Android free floating windows. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 759–770.

[26] Wu Zhou. 2016. https://www.fireeye.com/blog/threat-research/2016/06/latest-android-overlay-malware-spreading-in-europe.html, accessed on 09. June 2018. (2016).