Davide Bove* and Anatoli Kalysch

In pursuit of a secure UI: The cycle of breaking and fixing Android's UI

https://doi.org/10.1515/itit-2018-0023
Received August 23, 2018; revised January 24, 2019; accepted February 28, 2019

Abstract: Hijacking user clicks and touch gestures has become a common attack vector and offers a stealthy approach at escalating the privileges of a process without raising red flags among users or AV software. Exploits falling into this category are categorized as clickjacking attacks and have gained increased popularity on mobile devices, Android being the recent victim of a series of UI vulnerabilities.

Focusing on the Android OS this paper highlights previous and current UI-based attack vectors and finishes with an overview of security mechanisms, covering both system-wide as well as app-level protection measures.

Keywords: Android, Clickjacking, UI, Overlay, Security

ACM CCS: Security and privacy → Systems security → Operating systems security → Mobile platform security

1 Introduction

Clickjacking is an umbrella term referring to techniques of hijacking mouse clicks or touch gestures of users. Starting out on desktop devices, with a special emphasis on browser-based clickjacking attacks, touch hijacking recently found its way into mobile platforms.

While UI based attacks have been sighted on iOS and Windows Mobile [7, 14], recently the main focus seems to be shifted to Android. Different types of malware were discovered using UI-based attack vectors, including ransomware, adware, credential stealers and even accessibility-based UI rootkits, which are able to simulate whole user interactions [6, 8, 1]. Attackers often use clickjacking in conjunction with accessibility (a11y) services, either to enable an a11y service, to enable other permissions or to sniff sensitive information directly, such as banking data and 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. Combining clickjacking attacks with a11y services, a permission-less app can be used to bootstrap attacks, fully controlling the UI of a system. This works by neither requiring privilege escalation attacks, such as root exploits, nor requiring the user to consciously approve a11y services when running on Android versions prior to 8 [8].

Defenses against these attacks are being steadily developed by AOSP maintainers as well as external researchers. On Android the current research focus is centered around overlays, windows presented on top of the UI, and accessibility services due to their ability to simulate user interaction [13, 10, 8, 12, 16].

The main contribution of this paper is to raise awareness for the dangers of UI-based attacks on Android. First, we provide a detailed overview of UI based attacks on Android and their main attack vectors. Second, we show that the proposed defense mechanisms for app developers and AOSP maintainers were broken in incremental research. Finally we summarize which limitations apply to the current defensive mechanisms and which UI-based attack vectors remain open on Android to this day.

2 Android's security mechanisms

Android's security concept envisions a strong separation of app resources. An application level sandbox is enforced on resources to prevent unauthorized access to another app's data. User-granted permissions enforce control over system resources and exported app interfaces, which are verified on a per-request basis by the kernel driver "*binder*". This separation is even enforced in the UI, where every app has a *binder* token that determines which windows it is allowed to interact with. The following paragraphs describe the structure of Android's UI, and relevant actors.

2.1 User interface

Android's UI architecture needs to extend to a plethora of device groups with very diverse hardware profiles and

***Corresponding author: Davide Bove,** Friedrich-Alexander Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 1, Martensstr. 3, D-91058 Erlangen, Germany, e-mail: davide.bove@fau.de
Anatoli Kalysch, Friedrich-Alexander Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 1, Martensstr. 3, D-91058 Erlangen, Germany, e-mail: anatoli.kalysch@fau.de

screen sizes. Despite this variety the underlying UI model remains the same for every device and is build upon two major components: Views and Windows.

A **View** represents any visual element: a button, an image, or just a text element. The layout of the app, and thus the structure of a Window and its contained View objects, are defined by the developer. There are also View groups, which can contain other Views and define how they are visually aligned. Views and View groups are created in a hierarchical tree structure, with a *ViewGroup* object as root element and multiple View objects as leaves.

A **Window** is defined by an area on the screen where one or multiple Views are drawn. Typically, a Window instance is bound to an Activity and presented to the user as soon as the Activity becomes active. Window management is handled by the *WindowManager*. It is responsible for the UI presentation, e. g. fitting different windows to the current screen. There are different types of Window objects, which differ in size, color, content and display order.

In the context of clickjacking, transparency levels and the z-order are regularly leveraged to mount attacks. While the former defines an opacity value for the currently displayed Window, the latter defines in which order the Windows are drawn on the screen. A high z-order means that the Window is drawn on top of other Windows and constitutes an *overlay*. Android differentiates between different types of overlays that are used depending on the task at hand.

For example, a Toast is a little pop-up usually shown at the top or on the bottom of the screen. The intended use case is communicating short pop-up messages to the user, so it uses a high z-order value to be displayed on top of application windows. However, Toast windows can be set up to contain arbitrary content, such as buttons, media files or other UI elements. System windows are also displayed on top of everything else on the screen. They are designed to be used by the system, to show errors and alerts regarding critical information, but can be also used by third-party, non-system apps. Different from Toast messages, this requires the *SYSTEM_ALERT_WINDOW* permission.

Windows can further be modified by certain flags that are set on a per-window basis. One of the earliest UI security mechanisms is *FLAG_SECURE*, which prevents screenshots and screen recordings of a selected window. For overlay windows the two flags *FLAG_NOT_FOCUSABLE* and *FLAG_NOT_TOUCHABLE* allow modifications to the behavior of the overlay. An overlay can be declared as "pass-through", which means that it can not capture any touches (not touchable) and does not retain the focus (not focusable). If a user touches such an overlay, the touch is passed to the underlying window, and the app controlling the overlay is not notified that the touch happened. On the other side, if a touchable overlay is touched, the overlay registers the click, but will not propagate it to the underlying window. This is a security measure of Android, that prevents specific keylogging attacks where a malicious app uses an invisible fullscreen overlay to record the user's touch events. Such an attack could learn the user's passwords, any messages sent or the unlock pattern of the device.

2.2 Accessibility services

Accessibility services have a unique role in Android's UI. Designed to help users with disabilities they are given special capabilities to observe currently active UI elements and even perform UI interactions as if they were triggered by a real user. This includes even the generation of input and interaction with elements protected by the *secure flag*.

These abilities soon gained the attention of malware authors [13], who leveraged a11y services to gain Device Administration privileges or stole sensitive data and login credentials. To prevent abuse, Android introduced the permission *BIND_ACCESSIBILITY_SERVICE* which all a11y services need to request. Contrary to normal permissions, that are granted to the app upon installation, or dangerous permissions, which the user needs to grant during runtime, enabling an a11y service requires navigating through the settings of the device. The user is guided to the a11y settings where the service needs to be selected and explicitly turned on (with an additional confirmation dialog). Examples of confirmation dialogs are presented in Figure 1.

3 Progression of UI hijacking

Clickjacking enjoyed popularity in online and web-based venues, especially for browser-based exploitations, leading to research on practical attacks and defense methods. The first occurrence of clickjacking on the web reaches back to 2008, when Grossman and Hansen demonstrated a clickjacking attack involving Adobe's Flash Player, using transparent HTML elements to trick the user into giving webcam access to a Flash application [9].

Android's first app-based clickjacking attacks were analyzed around 2011. Johnson [11] discusses Toast messages as attack vectors and details in a proof of concept (PoC) how an attacker can leverage the high z-order of Toasts to overlay the currently active app. The PoC didn't

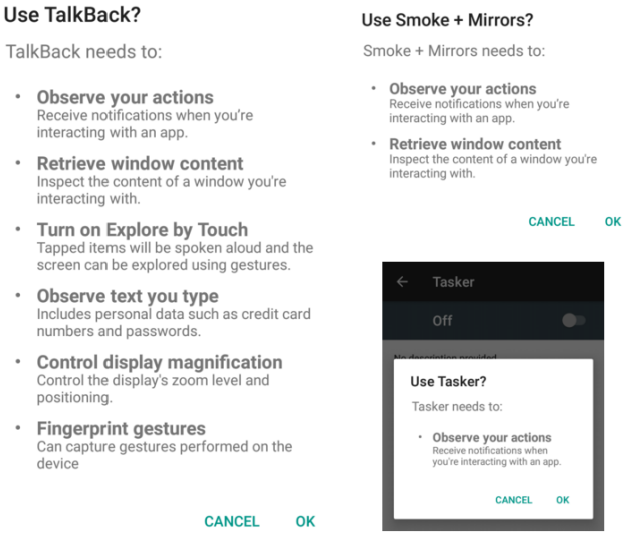

Figure 1: Confirmation dialogs presented in the last activation step for an a11y service [12]. The capabilities presented in this dialog are a direct result of the capabilities in the a11y service's configuration file.

require any permissions and allowed to transfer any taps and touches to the currently active window located directly below the Toast overlay. Since the Toast message could contain arbitrary contents, like pictures, buttons and various UI elements, this provided malware authors with a powerful tool. Similar attacks were identified in 2012, where existing attacks on the web were applied to mobile devices [15]. The relevance of such an attack was then demonstrated using a fullscreen overlay that covers the screen contents. The overlay placed an innocuous-looking button on top of the victim's Phone application, so the user would touch it and initiate a call to an expensive premium phone number, without his or her knowledge.

Over the years, naive clickjacking attacks have been improved through the use of newly discovered side-channels. Chen et al. [4] were the first to rely on the WindowManager's *FrameBuffer*, which is used to draw the UI, to infer additional information about other apps. Fernandes et al. [6] use a side-channel in the binder to query additional information about currently active applications.

Other approaches rely on the a11y framework to detect application launches [13], to steal user credentials and to partially circumvent Android's sandbox principle [10]. Fratantonio et al. [8] leverage the a11y framework and use overlays to install additional applications to create a greater foothold on the device. Additionally, they leveraged overlays themselves as side-channels by creating an overlay grid over the Android keyboard, and creating partial overlays to mitigate the *obscured flag*, a technique implemented in Android to mitigate clickjacking. If a touch passes through an overlay to an application, the obscured flag allows the app to acknowledge this by checking its value. Bianchi et al. [3] identified additional attack vectors with overlays. Besides pass-through overlays, which pass touches on to the underlying UI, they also leverage overlays that trap touches and gestures, rendering a device inaccessible.

Initial attempts from malware at abusing overlays focused on privilege escalation. In 2015, the malware family *Android/BadAccents* was discovered to be using clickjacking techniques to obtain Device Administrator privileges [17]. Rasthofer et al. [18] analyzed 263,623 applications from the Google Play Store and found over 99.96 % applications to be vulnerable to the same overlay attack vector.

Newer attempts focus strongly on credential and banking information leakage by using overlays to simulate the login Activity of apps. Side-channels enhancing these techniques have been encountered as well, like the use of GPS location information in combination with deep links inside apps [21]. In the beginning of 2018, samples of the *Android/FakeApp* family displayed this behavior to simulate a successful login into the Uber app and to display a map with the current location.

4 Mitigation approaches

There are several ways to use UI attacks to compromise the device, from logging touches to installing applications in the background, without the user noticing. There are different approaches to minimize the damage of malware, ranging from protections for third-party apps to pervasive changes to the underlying operating system. We differentiate between three types of mitigations:

- **System-level modifications** change the underlying operating system, so you have a completely different version of the Android OS. This usually requires physical access to the device and a procedure called *flashing* to use it.
- **Root-level modifications** require a device with elevated – so-called root – privileges. Some technical knowledge is required to root a device, so the method is not applicable for every Android user.
- **Application-level modifications** are implemented on a per-app basis and do not require any changes to the system. Every app needs to implement their own protection, but can be deployed on devices more easily.

We also discuss protections implemented by the Android team across the different OS versions.

4.1 Touch filtering

The Android framework provides only a single concept that can be used by third-party developers to mitigate clickjacking attacks by malicious applications. It is called Touch Filtering and is a security mechanism that was introduced with the release of Android 2.3. Through Touch Filtering, the system discards touch events on a View if another window is obscuring it. Developers can enable this feature for specific layout elements, effectively disabling the use of an app if an overlay is on top of it. Combined with *FLAG_WINDOW_IS_OBSCURED*, a special flag for touch events, the app can react to the touch of a covered UI element, and display a warning. As an example, a protected button can be used like any other button as long as there is no window on top of the app's Activity. It is clickable and it executes any action associated with it. If a window is on top, be it an invisible Toast message or simply a system dialog, the button becomes unusable. Related work makes use of this technique to protect an Activity and specifically block clickjacking attempts [15].

The problems with this method are manifold. First, it is not enabled by default and as a developer, you need to enable it for every single View, one by one. That is not practical for existing applications and the Android documentation does not really explain which views should be protected and why. Second, even though it protects against simple clickjacking attacks, where a fake button is placed over a real (protected) button, it offers no real protection against sophisticated attacks, where users are tricked into clicking a view without having an overlay covering the view. An instance of this is described in related work, where multiple overlays are used to create a hole around a button [8]. In such a setup, the touch filter would not be triggered and no overlay would be detected by the activity.

A further problem is that there are legitimate applications on the market that make use of (fullscreen) overlays. An example is Twilight, an app with over 5 million installs that activates a "night mode" for the screen. The app uses a semi-transparent overlay that spans over the whole screen and is always on top of other activities. Apps which enable Touch Filtering on single views make these views unusable as long as the app is active. The main problem with this is that users are not notified by default about why the app they want to use is not reacting to touches, and might think it is broken. The developer has to react to this specific event and check the obscured flag, then show an appropriate warning. In a recent study, 263.623 apps selected from different categories of the Google Play Store were analyzed [18]. The researchers found that only 369 (0,13 %) of the apps were using Touch Filtering, concluding that developers are "not aware about this attack vector" and consequently do not implement it.

The same method is used for specific system apps in Android 6.0 and below. The `Package Installer` app, which is used when new apps are installed outside of the Play Store, implements it. Through the *Install* button, the screen is checked for overlays and the app shows a notification when it detects an overlay ("Screen overlay detected!"), warning the user that it should remove the overlay manually, by closing the app (but not specifying which app causes the problem).

4.2 Modifying the system

The state-of-the-art clickjacking techniques on Android involve overlay windows. Overlays, also called *floating windows*, are a special kind of UI elements. Since they float over other windows and you can have multiple windows on the screen, they allow multitasking, similar to desktop applications on a regular computer. This is a novelty on Android devices, or at least it was until native support for split-screen mode was implemented in Android 7.0 Nougat. Still, floating windows are a popular feature for devices with bigger screens, such as tablets. Figure 2 shows the screenshot of such a popular app, which uses overlays to enable multitasking windows.

So, not only is it difficult to mitigate malicious overlay usage, but it is also a challenge to differentiate between le-

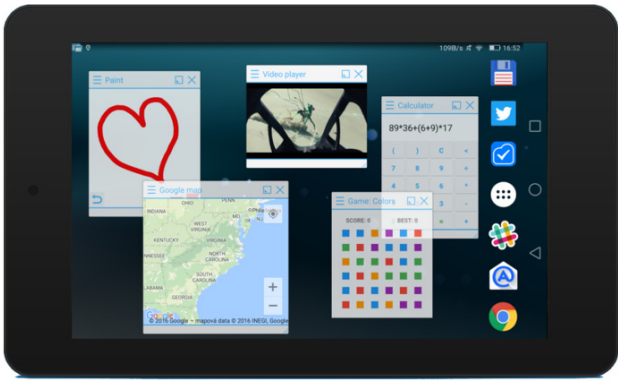

Figure 2: Screenshot of the "Floating Apps" application, which has over 1 million installs on the Play Store.

gitimate uses of the API and apps that abuse the feature. So how can such a malign app be recognized before an actual attack occurs?

To find a solution, it is crucial to be able to distinguish between benign and malicious uses. While it is known at install time if an app has the permission to use overlays, it is not known if the app will use them at all. Besides antivirus apps that can analyze an app's information to determine if it corresponds to a known malware, there are approaches that use static analysis to detect potential GUI attacks [3]. Static analysis refers to the analysis of software without executing it, and it can usually catch malicious behavior that is not hidden by encryption or obfuscation techniques. More specifically, this approach looks for suspicious API calls at the byte-code level, such as the `WindowManager.addView` method. This is done by analyzing the APK file, the packaged application file format of Android.

While the static analysis approach can detect all apps that use specific procedures, the detection of benign apps that use the same API is only partially solved. In such cases, the user needs to be included in the decision, so the approach only offers a pre-filtering of potential apps. Deciding which application is currently on the screen is not easy for a user. The possibility of malicious apps to pass off as parts of a legitimate application is high, as the possibilities to show dialogs and overlays has only few limitations on Android. On the system level, this can be easily solved introducing security indicators as a modification of the operating system [3].

Another interesting concept is the use of an "Overlay Mutex" [6]. A mutex (mutually exclusive) object is often used in parallel programs to synchronize the access to a resource, by allowing only one instance or thread to access the resource at a time. Similar to that, an Overlay Mutex makes sure that no app other than the active one can put an overlay window on top of the screen. If a background app attempts to start an overlay, a notification is shown that warns the user. This can effectively prevent a clickjacking attack, if used correctly, but it can also break existing applications that permanently show an overlay, making them unusable. Visually, a lock icon or a similar indicator has to be shown on the screen most of the time, which is often not compatible with Android's customization principles. Such visual indicators, inspired by the HTTPS lock icons found in browsers, have their own share of problems and considerations regarding usability and comprehension [20, 5]. The Overlay Mutex approach is effective and simple to use, yet it only offers protection against a single class of attacks and needs to be combined with other methods to filter out benign apps.

While the previous approaches to detection and mitigation of clickjacking attempts aimed at modifying the operating system, there are also proposals for extending the Android SDK. Therefore, there are only minor changes to the system itself, but developers can use them to protect their apps. Similar to the `FLAG_SECURE` flag that prevents taking screenshots or recordings of a window, there is also the possibility to extend the developer API and provide a `setSecureWindow` method to mark windows as sensitive. Then, the system can easily identify the window and fire an `onOverlap` event whenever an overlay is on top of it. Therefore, the developer can prepare for such an event and react accordingly, showing a warning or disabling sensitive UI controls [2]. This method can not reliably distinguish between a malicious and a benign overlay, so it us up to the developer to take a decision.

While this is the least opinionated solution of all the approaches in this chapter, this method suffers from the same adoption and distribution problems of the other ones. Additionally, the decision to include the developer into the protection process can be both good and bad. On the one side, developers can easily implement a protection for their app, which is good for the end user. On the other side, comparing it to [3] and [6], the method itself does not benefit the user in any way, as long as developers do not update their apps and implement effective countermeasures.

The last work of this section describes a defense scheme that does not require the user for detection. As the goal is to detect malicious overlays, the detection algorithm can be based on the following four assumptions [22]:

- The app of the overlay is different from the receiver app of the touch input.
- The window type of the overlay has a higher z-order value.
- The overlay is pass-through and uses the flag *FLAG_NOT_TOUCHABLE*.
- The overlay's transparency (*alpha*) attribute is above 95 %.

The detection approach is similar to the other approaches: all overlays on top of an Activity are analyzed, one by one. The analysis works by monitoring calls to the relevant parts of the Window management process, e. g. opening and closing windows, showing and hiding an Activity. When such a suspicious overlay is found, the user is given the option to uninstall the offending app, such that the user has no control over the detection, but still has the final say over what to do with an app.

4.3 Working with root privileges

For users with rooted devices, making changes that affect the system is easier. Xposed is a framework for rooted Android devices that uses code injection to change the system's behavior, such that no manual changes to the OS source code is required. With Xposed, users can use modules that are provided by a community of independent developers and can be installed like regular mobile apps.

An approach called "Android Window Integrity" (AWI) makes use of the Xposed framework [19]. It is a security concept that protects the user against a specific class of UI attacks, namely *Window overlay* and *Task hijacking* attacks. While the former category indicates the class of attacks described in this work, the latter category refers to manipulations of the Activity stacks and of the back-button behavior. The AWI concept focuses on reimplementing the logic behind the Android navigation, called *Back Stack*, consisting of Activity stacks and Window management. At the center of this is the notion of "activity sessions", which are records of the actual sequence of views shown to the user. These sessions are bound to one specific app, and usually begin with the launcher activity, which is the first activity to be shown when the user opens the app. If this sequence of views, a view being any Window that was or is still visible to the user, is different from Android's own back stack sequence, a suspicious behavior is assumed. Also, the system checks if all views shown to the user in one activity session are from the same app. The rationale is that an app is considered suspicious if it manipulates the back stack and creates a window without user interaction. When that is the case, the user is notified of the problem and is required to take a decision on how to react. The alert which is shown explicitly asks the user if he or she wants to block the window and shows the application package name to identify the app. This behavior is similar to most popup blockers implemented in browsers, which first ask for confirmation before showing the potentially offending element.

The actual implementation of AWI by the authors is called "WindowGuard" and written as a module for the popular Xposed framework. The authors argue that this allows WindowGuard to be deployed on a wide range of devices. In the evaluation of their work, WindowGuard was tested on the 12,060 most popular apps from the Google Play Store, which resulted in 1.03 % of apps triggering the alerts. As the authors explain, the alerts were triggered by floating windows that had different purposes: from controlling a music player to showing ads. Also, the performance evaluation implies that the impact and overhead of WindowGuard is minimal (0.45 %), even though the authors did not consider the overhead of the Xposed framework itself, which is required for their solution. Overall, the AWI model seems appropriate for detecting most clickjacking attacks on Android. While the work does not go into detail on how it performs on different types of overlays, the described approach seems to work irrespectively of how the windows are created.

There are some minor issues that need to be addressed for this paper. First, relying on user interaction and decision to implement security is a double-edged sword. In this case, WindowGuard shows the package name in the alert, which can be easily spoofed by malicious applications. It is not a reliable identity proof for an app, as package names can always contain keywords that do not match the app's launcher name. Also, if a malicious application repeatedly attempts to show an offending window, a blocking alert is shown every time, which might annoy the user and may lead to the uninstalling of WindowGuard. This reaction, called *alarm fatigue*, can be expected if the user assumes that the error is caused by a malfunctioning of the module. In addition to that, if the malicious app is explicitly launched by the user, WindowGuard does not block any windows. This could be the case when the user is tricked into opening what appears to be a benign app. This includes phishing, spoofing and regular social engineering attacks. While this is unfortunate, it is not a specified goal of the paper. Instead, the usability of the security measure could be analyzed with an extensive study to solve this issue.

4.4 Using and abusing the Android SDK

In this section, we discuss a practical example to show how the existing Android SDK and its limited API can be used for different purposes than originally intended. Also, this can be implemented more easily on different devices than previous solutions, as the method does not require changes to the system.

The approach is called "Window Punching" and can be completely implemented on the application level, without modifications to the system [2]. The technique involves sending touch events to an application and observing which of these events are received by the app. On the receiver side, which is the application using the protection, the app can determine from the received events if there is another window between the user and the application. The relevant settings for this technique are the number of events and the position. As seen in Figure 3, either the whole screen or only specific areas of it can be "punched".

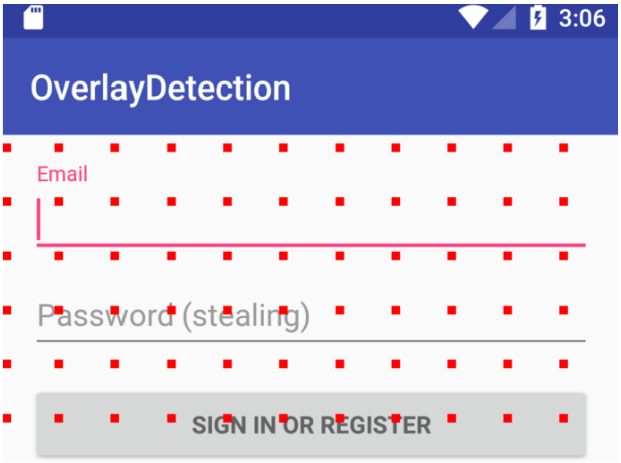

Figure 3: Example of Window Punching technique where every dot corresponds to a simulated touch detected by the protected application.

In the first case, one can use a grid-like setup to systematically scan the screen, pixel for pixel. Scanning means sending a virtual touch event to the Activity window, which is detected by the application. The detection is given by the `MotionEvent` API, which gives detailed information about the interaction of the user with the app. Similar to Touch Filtering, this method detects if there was any overlay obscuring the app at the time of the touch event.

The process of simulating touch events requires some processing power, doing so on every pixel on the screen repeatedly is therefore not practical on most mobile devices. A strategy to this problem is scanning only the critical parts of the activity, for example buttons and text input fields. Also, we may need to calculate the optimal grid size for the scan. Using a grid with more space between two points results in less events to generate and can be executed faster, but we could miss some smaller overlays. On the other side, a very dense grid is able to detect any overlay, but requires much more computing power. So, a sensible trade-off between performance and detection accuracy is needed.

The ability to generate custom touch events is granted by the *Instrumentation* library of the Android SDK, which does not require any special permissions to use. In general the library is intended for instrumented unit tests, which are used for automated testing of the UI. In such a test, a developer can decide which elements to interact with and navigate through an app without user interaction, in order to test the functionality of the app. Through the provided methods, a developer can send custom *MotionEvent* messages to the app. Such events are usually generated by the system whenever a user touches or interacts with the touch screen, thus an instance contains information about the type of touch (using a finger, a touch pen etc.) and the coordinates of the event.

The most important limitation of this is that an app can only instrument itself, but not other apps. This limitation can be utilized to detect windows that catch the input, such as during phishing attempts: Whenever a touch event is fired and hits a window not owned by the current app, a *SecurityException* is thrown in the Instrumentation code. This is the case for malicious floating windows, which are put on top of the screen and catch the input focus. During phishing attacks, for example, the user usually enters sensitive data into a field the attacker controls. If such a focusable window is put on top of an app implementing the protection, the app triggers the exception above and can react accordingly, by warning the user and raising awareness of what is really happening.

The above method is a smart and easy way to protect a single app from phishing attacks using overlays. The difference to other methods is that developers can implement, tweak and control the protection measure themselves. As described above, by combining the touch simulation feature with Touch Filtering and tweaking the parameters for optimal user experience, developers can achieve a pretty effective application-level security countermeasure.

4.5 Protection mechanisms by Google

Previous sections discussed techniques that would allow application developers or security-conscious users to prevent abuse. They either needed to be adapted on a per-application basis or required changes to the Android system itself. Meanwhile, Google introduced protection measures to mitigate abuse through UI-based attacks.

Android 5.0 Lollipop saw the introduction of security mechanisms specifically aimed at clickjacking. New code was introduced into some system apps that prevent the display of windows on top of selected critical system dialogs (such as the Device Admin confirmation or the permission settings). In another case, for Accessibility services, Touch Filtering was enabled for some elements. For the confirmation dialog, the “OK” button is protected by the *setFilterTouchesWhenObscured* method. A year later, the Touch Filtering method was replaced by the obscured flag, showing a warning message if any overlays were detected on top of the button.

Furthermore, since Android 6.0 Nougat, the Android team has assigned a higher priority to the *SYSTEM_ALERT_WINDOW* permission, requiring the user to

explicitly grant this permission through the Settings app. An exception is made for apps installed from the Play Store, which are granted this dangerous permission by default.

While these measures protected against most basic clickjacking attacks, a new approach shows how they still can be circumvented [8]. Through combination of the above permission and an accessibility service, the device can be taken over easily, including the deployment of malware with full device permissions (so-called "God-mode" app). The attack relies on two basic design flaws: the first is that apps installed from the Play Store do not need to be granted the "draw on top" permission, since it is automatically granted at install time. The second issue is that, as the authors claim, it is trivial to upload and distribute such an app through the Play Store, since the automatic scans done by Google do not seem to reliably catch features like code side-loading and the abuse of overlays. Using these facts and an elaborate clickjacking technique, the user can be fooled into enabling an Accessibility service that allows the execution of further attacks. The clickjacking technique can be considered a novelty, since it works despite the security measures introduced by Google, namely Touch Filtering and the obscured flag.

The two concepts are called *Context-aware clickjacking* and *Context hiding*. Context-aware clickjacking occurs when the app performing the attack knows what is being clicked and reacts with relevant and reasonable output. Context hiding describes a technique used to hide the underlying "privileged" application (e. g. the Settings app, or the Permissions dialog), tricking the user into interacting with it.

The findings of *Cloak & Dagger* [8] lead to a new, undocumented security measure by Google, implemented in Android 7.1.2. The code, as shown in Figure 4, uses the *AppOpsManager* class, which can only be used by system apps, to block any overlays from showing while the

```
public static void setOverlayAllowed(Context
    context, IBinder token, boolean allowed) {
    AppOpsManager appOpsManager = context.
    getSystemService(AppOpsManager.class);
    if (appOpsManager != null) {
        appOpsManager.setUserRestriction(
    AppOpsManager.OP_SYSTEM_ALERT_WINDOW,
                !allowed, token);
        appOpsManager.setUserRestriction(
    AppOpsManager.OP_TOAST_WINDOW,
                !allowed, token);
    }
}
```

Figure 4: Method used to block any overlays in system apps on Android 7.1.2.

Accessibility confirmation dialog is shown. In addition to that, the flag *FLAG_WINDOW_IS_PARTIALLY_OBSCURED* was introduced. In contrast to the regular obscured flag, this new flag is set if the touched button has any overlaying window on top, regardless of how and where the user touches it.

With the release of Android 8.0 Oreo, Google made a few significant system changes that affect overlays. By deprecating all previous overlay types and introducing the new window type *TYPE_APPLICATION_OVERLAY*, apps can only create overlays that display over other activities, but can not cover "critical" system windows, such as the status bar or the keyboard. In addition, to protect system apps against future overlay attacks, the Android team introduced the new system-only permission *HIDE_NON_SYSTEM_OVERLAY_WINDOWS* and a new Window flag, which is appropriately called *PRIVATE_FLAG_HIDE_NON_SYSTEM_OVERLAY_WINDOWS*. When a window with this flag is active, all overlays created by any other app are hidden from the screen. The overlays only reappear when the window is removed or hidden again. This is implemented for some critical confirmation dialogs to prevent Context Hiding.

5 Limitations

The proposed mitigations against clickjacking and UI related attacks have severe limitations regarding availability and usability. Most academic protections presented in this paper have a common issue: They require in-depth changes to the OS, either by using a modified Android version or by rooting the device. This again requires compatible devices and some technical knowledge to achieve. Although these are valid solutions, they are not practical as long as they can not be easily implemented by and for regular users. Also, persuading Google or other OS providers (OnePlus, LineageOS etc.) to integrate these solutions is tedious and difficult to achieve.

Another issue is that, due to the widespread use of Android in multiple versions and flavors, the suggested changes would not be applied to most devices, as Android suffers from version fragmentation and a number of unsupported and outdated devices on the market. It is therefore unrealistic that such changes ever make it into an official version by Google, or are implemented by any third-party OS provider. Therefore, even if developers would need to update their software, application-level measures are preferable.

Since Android 8.0, the new countermeasures mitigate some of the clickjacking techniques presented in this paper. With Context Hiding, the user is still easily fooled into clicking any elements in the Settings app, which is not protected by the private window flag. But at least the critical confirmation dialogs can not be hidden or manipulated by overlay windows anymore.

6 Conclusion

After iterating over the basics of Android UI components, section 2 listed the different clickjacking techniques on Android. As a contribution, this paper summarized in section 4 the several potential approaches to mitigate clickjacking attacks, from app-based methods to system-level modifications. Most solutions require in-depth manipulations of the Android source code, changing the overall design of Android, while others could be applied by more or less technical users. Touch Filtering, the obscured flag and some internal protections were presented as the major countermeasures introduced by Google, showing that protection against UI attacks on Android has become more than just an afterthought.

While the latest security features are being deployed on new devices, a majority of devices is still vulnerable to these attacks. This is due to the nature of Android's ecosystem, and the refusal of device manufacturers to provide security updates for older devices. Also, most effective countermeasures were reserved for system apps, disallowing the use for third-party applications. It is thereby crucial that developers are given access to security-related APIs to add their own protection measures into their apps. On the other side, one can see that current defenses against overlay attacks are ineffective and therefore unused by third-party apps.

The biggest problem with the current state of overlay mitigation is that, even though there are working protections, they are exclusive to system apps and can not be used by third-party apps and developers [12]. This means that every other app, and especially apps with sensitive information (messaging, banking or dial apps), are still vulnerable to clickjacking attacks. It is up to the Android developers to decide what windows are protected. As of today, only a number of critical dialogs implement the countermeasure. Users can still be tricked into enabling system settings that are not protected, such as the developer settings. As a consequence, the Android team decides on a per-window basis which parts of the OS are "critical" enough to receive the protection.

We argue that this approach is flawed, as any setting included in the Settings app can be abused by an attacker to further deceive the user. As Android does not allow to change any system settings programmatically, an app should not be able to circumvent this by tricking the user. Of course blocking overlays entirely in Settings would break some apps' functionality, but it would also signal a clear commitment to security by the Android team.

Acknowledgment: We thank Prof. Dr.-Ing. Freiling and Tobias Groß for their helpful comments on earlier versions of this paper.

Literature

1. Vitor Afonso, Anatoli Kalysch, Tilo Müller, Daniela Oliveira, André Grégio, and Paulo Lício de Geus. Lumus: Dynamically uncovering evasive Android applications. In *International Conference on Information Security*, pages 47–66. Springer, 2018.
2. Abeer AlJarrah and Mohamed Shehab. Maintaining user interface integrity on Android. In *Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual*, volume 1, pages 449–458. IEEE 2016.
3. Antonio Bianchi, Jacopo Corbetta, Luca Invernizzi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. What the app is that? Deception and countermeasures in the Android user interface. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 931–948. IEEE, 2015.
4. Qi Alfred Chen, Zhiyun Qian, and Zhuoqing Morley Mao. Peeking into your app without actually seeing it: UI state inference and novel Android attacks. In *USENIX Security Symposium*, pages 1037–1052, 2014.
5. Adrienne Porter Felt, Robert W Reeder, Alex Ainslie, Helen Harris, Max Walker, Christopher Thompson, Mustafa Embre Acer, Elisabeth Morant, and Sunny Consolvo. Rethinking connection security indicators. In *SOUPS*, pages 1–14, 2016.
6. Earlence Fernandes, Qi Alfred Chen, Justin Paupore, Georg Essl, J Alex Halderman, Z Morley Mao, and Atul Prakash. Android UI deception revisited: Attacks and defenses. In *International Conference on Financial Cryptography and Data Security*, pages 41–59. Springer, 2016.
7. Lorenzo Franceschi-Bicchierai. The iPhone's constant password popups are a hacker's dream, may 2017. https://motherboard.vice.com/en_us/article/ne7gxz/ios-iphone-password-phishing-app-popups, accessed on May 29th, 2018.
8. Yanick Fratantonio, Chenxiong Qian, Simon P Chung, and Wenke Lee. Cloak and dagger: from two permissions to complete control of the UI feedback loop. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 1041–1057. IEEE, 2017.
9. Jeremiah Grossman. Clickjacking: Web pages can see and hear you, Oct 2008. http://blog.jeremiahgrossman.com/2008/10/clickjacking-web-pages-can-see-and-hear.html, accessed on April 20, 2018.

10. Yeongjin Jang, Chengyu Song, Simon P. Chung, Tielei Wang, and Wenke Lee. A11y attacks: Exploiting accessibility in operating systems. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS '14*, pages 103–115, ACM, New York, NY, USA, 2014.
11. Ken Johnson. Revisiting Android tapjacking, May 2011. https://web.archive.org/web/20171121203845/https://nvisium.com/blog/2011/05/26/revisiting-android-tapjacking/, accessed on June 1st, 2018.
12. Anatoli Kalysch, Davide Bove, and Tilo Müller. How Android's UI security is undermined by accessibility. In *Proceedings of the 2nd Reversing and Offensive-oriented Trends Symposium, ROOTS*, pages 2:1–2:10, ACM, New York, NY, USA, 2018.
13. Joshua Kraunelis, Yinjie Chen, Zhen Ling, Xinwen Fu, and Wei Zhao. On malware leveraging the Android accessibility framework. In *International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*, pages 512–523. Springer, 2013.
14. Tongbo Luo, Xing Jin, Ajai Ananthanarayanan, and Wenliang Du. Touchjacking attacks on web in Android, iOS, and windows phone. In *International Symposium on Foundations and Practice of Security*, pages 227–243. Springer, 2012.
15. Marcus Niemietz and Jörg Schwenk. UI redressing attacks on Android devices. *Black Hat Abu Dhabi*, 2012.
16. Andrea Possemato, Andrea Lanzi, Simon Pak Ho Chung, Wenke Lee, and Yanick Fratantonio. Clickshield: Are you hiding something? Towards eradicating clickjacking on Android. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18*, pages 1120–1136, ACM, New York, NY, USA, 2018.
17. Siegfried Rasthofer, Irfan Asrar, Stephan Huber, and Eric Bodden. How current Android malware seeks to evade automated code analysis. In *IFIP International Conference on Information Security Theory and Practice*, pages 187–202. Springer, 2015.
18. Siegfried Rasthofer, Irfan Asrar, Stephan Huber, and Eric Bodden. An investigation of the Android/BadAccents malware which exploits a new Android tapjacking attack. Technical report, TU Darmstadt, Fraunhofer SIT and McAfee Mobile Research, 2015.
19. Chuangang Ren, Peng Liu, and Sencun Zhu. Windowguard: Systematic protection of GUI security in Android. In *Proc. of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2017.
20. Stuart E Schechter, Rachna Dhamija, Andy Ozment, and Ian Fischer. The emperor's new security indicators. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 51–65. IEEE, 2007.
21. Dinesh Venkatesan. Android malware steals uber credentials and covers up the heist using deep links, 2018. https://www.symantec.com/blogs/threat-intelligence/android-malware-uber-credentials-deep-links, accessed on May 23rd, 2018.
22. Longfei Wu, Benjamin Brandt, Xiaojiang Du, and Bo Ji. Analysis of clickjacking attacks and an effective defense scheme for Android devices. In *Communications and Network Security (CNS), 2016 IEEE Conference on*, pages 55–63. IEEE, 2016.

Bionotes

M. Sc. Davide Bove
Friedrich-Alexander Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 1, Martensstr. 3, D-91058 Erlangen, Germany
davide.bove@fau.de

M. Sc. Davide Bove is a Master's graduate from Friedrich-Alexander University Erlangen-Nürnberg (FAU). He graduated in the field of Software Engineering and now focuses his studies on secure software, Android security and distributed networks.

M. Sc. Anatoli Kalysch
Friedrich-Alexander Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 1, Martensstr. 3, D-91058 Erlangen, Germany
anatoli.kalysch@fau.de

M. Sc. Anatoli Kalysch is a PhD student at Friedrich-Alexander University Erlangen-Nürnberg (FAU). His research interests include reverse engineering and program analysis, obfuscation techniques, and Android security. Anatoli Kalysch has a M. Sc. in computer science from FAU.