Cloak and Dagger: From ~~Two~~ Permissions to Complete Control of the UI Feedback Loop

(or, "Why Boring UI Bugs Matter")

Yanick Fratantonio
UC Santa Barbara & EURECOM
yanick@cs.ucsb.edu

Chenxiong Qian, Simon P. Chung, Wenke Lee
Georgia Tech
qchenxiong3@gatech.edu
pchung34@mail.gatech.edu
wenke.lee@gmail.com

NOTE TO THE READER

A first version of this work was published in the Proceedings of the IEEE Symposium on Security and Privacy 2017. This extended white paper, prepared for Black Hat USA 2017, presents more details about the responsible disclosure of the design issues uncovered by this work and the reception of this work from the community and from Google (Section XI). We also present extensions of these attacks that show how they can be boostrapped from an app requiring one permission (for Android 7.1.2, see Section XIII) or even from an app requiring zero permissions (for Android 6.0.1 and earlier, see Section XIV). We also comment on how the upcoming version of Android (Android O), currently released as preview for beta testers, is affected by these attacks (Section XV). Finally, we conclude this paper with a few closing remarks and we urge Google to take action to secure the Android UI. Up-to-date information and demos can be found on our project website, http://cloak-and-dagger.org.

***Abstract*—The effectiveness of the Android permission system fundamentally hinges on the user's correct understanding of the capabilities of the permissions being granted. In this paper, we show that both the end-users and the security community have significantly underestimated the dangerous capabilities granted by the SYSTEM_ALERT_WINDOW and the BIND_ACCESSIBILITY_SERVICE permissions: while it is known that these are security-sensitive permissions and they have been abused individually (e.g., in UI redressing attacks, accessibility attacks), previous attacks based on these permissions rely on vanishing side-channels to time the appearance of overlay UI, cannot respond properly to user input, or make the attacks literally visible. This work, instead, uncovers several design shortcomings of the Android platform and shows how an app with these two permissions can completely control the UI feedback loop and create devastating attacks. In particular, we demonstrate how such an app can launch a variety of stealthy, powerful attacks, ranging from stealing user's login credentials and security PIN, to the silent installation of a God-mode app with all permissions enabled, leaving the victim completely unsuspecting.**

To make things even worse, we note that when installing an app targeting a recent Android SDK, the list of its required permissions is not shown to the user and that these attacks can be carried out without needing to lure the user to knowingly enable any permission. In fact, the SYSTEM_ALERT_WINDOW permission is automatically granted for apps installed from the Play Store and our experiment shows that it is practical to lure users to *unknowingly* grant the BIND_ACCESSIBILITY_SERVICE permission by abusing capabilities from the SYSTEM_ALERT_WINDOW permission. We evaluated the practicality of these attacks by performing a user study: none of the 20 human subjects that took part of the experiment even suspected they had been attacked. We also found that it is straightforward to get a proof-of-concept app requiring both permissions accepted on the official store.

We responsibly disclosed our findings to Google. Unfortunately, since these problems are related to design issues, these vulnerabilities are still unaddressed. We conclude the paper by proposing a novel defense mechanism, implemented as an extension to the current Android API, which would protect Android users and developers from the threats we uncovered.

I. INTRODUCTION

One of the key security mechanism for Android is the permission system. For the permission system to actually improve security, the end-users and the community need to be aware of the security implications of the different permissions being requested. In this paper, we focus our attention on two specific permissions: the SYSTEM_ALERT_WINDOW and the BIND_ACCESSIBILITY_SERVICE permissions. The former allows an app to draw overlays on top of other apps, while the latter grants an app the ability to discover UI widgets displayed on the screen, query the content of these widgets, and interact with them programmatically, all as a means to make Android devices more accessible to users with disabilities.

Even though the security community (as well as our adversaries) are beginning to discover the threats from the SYSTEM_ALERT_WINDOW and the BIND_ACCESSIBILITY_SERVICE permissions, we show how seemingly innocuous design choices can lead to even more powerful attacks. Moreover, we uncover how these two permissions, when combined, lead to a new class of stealthy, very powerful attacks, which we called "cloak and dagger"

attacks.[1] Conceptually, "cloak and dagger" is the first class of attacks to *successfully and completely compromise the UI feedback loop.* In particular, *we show how we can modify what the user sees, detect the input/reaction to the modified display, and update the display to meet user expectations. Similarly, we can fake user input, and still manage to display to the user what they expect to see, instead of showing them the system responding to the injected input.*

This is in sharp contrast to existing attacks that utilize only *one* of the SYSTEM_ALERT_WINDOW and BIND_ACCESSIBILITY_SERVICE permissions. With only SYSTEM_ALERT_WINDOW permission (e.g., GUI confusion attacks [1], [2], [3]), the attacker can modify what the user sees, but cannot anticipate how/when the user reacts to the modified display, and thus fails to change the modified displayed content accordingly. Similarly, with only BIND_ACCESSIBILITY_SERVICE permission, the attacker can inject fake user inputs[2], but the attacker cannot prevent the user from seeing the results of these fake inputs displayed on the screen. As such, in both cases, *with only one of the two permissions, the user will very quickly discover the attack*. On the contrary, in "cloak and dagger," the synergy of the two permissions allows an attacker to both modify what the user sees and inject fake input, *all while maintaining the expected "user experience" and remaining stealthy.* Such stealthiness would in turn lead to better sustainability (i.e., the malicious app can be made available on the Play Store and remain there for a very long time).

We will also demonstrate the devastating capabilities the "cloak and dagger" attacks offer an adversary by showing how to obtain almost complete control over the victim's device. In this paper, we will demonstrate how to quietly mount practical, context-aware clickjacking attacks, perform (unconstrained) keystroke recording, steal user's credentials, security PINs, and two factor authentication tokens, and silently install a God-mode app with all permissions enabled. We note that by completely controlling the feedback loop between what is displayed on screen and what is inputted by the user, "cloak and dagger" attacks invalidate a lot of security properties that are taken for granted (e.g., the user will eventually notice something and take action), and make the uncovered design issues more dangerous.

What makes "cloak and dagger" attacks even more dangerous is the fact that the SYSTEM_ALERT_WINDOW permission is automatically granted for apps installed from Play Store, and it can be used to quietly lure the user to grant the BIND_ACCESSIBILITY_SERVICE permission and bootstrap the whole attack. Furthermore, it is straightforward to get a proof-of-concept app requiring both permissions accepted on the official store.

To test the practicality of these attacks, we performed a user study that consisted of asking a user to first interact with our proof-of-concept app, and then login on Facebook (with our test credentials). For this experiment, we simulated the scenario where a user is lured to install this app from the Play Store: thus, SYSTEM_ALERT_WINDOW is already granted, but BIND_ACCESSIBILITY_SERVICE is not. The results of our study are worrisome: even if the malicious app actually performed clickjacking to lure the user to enable the BIND_ACCESSIBILITY_SERVICE permission, silently installed a God-mode app with all permissions enabled, and stole the user's Facebook (test) credentials, *none* of the 20 human subjects even suspected they have been attacked. Even more worrisome is that none of the subjects were able to identify anything unusual even when we told them the app they interacted with was malicious and their devices had been compromised.

We reported our findings to Google, which promptly acknowledged all the problems we have raised. However, no comprehensive patch is available yet: while few of the specific instances of problems can be fixed with a simple patch, most of the attacks are possible due to design shortcomings that are not easily addressable.

We conclude this paper by elaborating on a new, principled defensive mechanism that would prevent, by design, the presence of this new class of attacks. In particular, the protection system involves an extension of the Android API, in a way that would allow the developer to indicate to the OS that a given widget plays a security-sensitive "role." In turn, when any of these widget is displayed, the OS would enforce, in a centralized manner, a number of constraints on the capabilities of third-party apps with respect to the two permissions we discuss in this work.

While we believe our proposed defense system would significantly improve the overall security of the Android platform, the current state of Android security patch distribution would currently leave most devices unpatched [5], and thus susceptible to the problems uncovered in this work. Thus, we hope this work will urge Google to reconsider their decision of automatically granting the SYSTEM_ALERT_WINDOW permission to apps hosted on the Play Store: This modification could be quickly deployable as it only affects the Play Store app itself. To the best of our knowledge, Google is refraining to deploy this fix because this permission is requested by top apps installed by hundreds millions of users (e.g., Facebook, Messenger, Twitter, Uber), and the new permission prompt would interfere with the user experience. While these concerns are understandable, we believe users and the security community should be able to make informed decisions.

In summary, this paper makes the following contributions:

- We uncover several design shortcomings related to the SYSTEM_ALERT_WINDOW permission, the BIND_ACCESSIBILITY_SERVICE permission, and the Android framework itself.

[1]The term "cloak and dagger" can refer to: 1) situations involving intrigue, secrecy, espionage, or mystery; or 2) in martial arts, literally wielding a dagger in one hand and a cloak in the other. The purpose of the cloak was to obscure the presence or movement of the dagger, to provide minor protection from slashes, to restrict the movement of the opponent's weapon, and to provide a distraction.

[2]E.g., in [4], the BIND_ACCESSIBILITY_SERVICE permission is abused to inject click events to allow the installation of adware and other unwanted apps.

- We show that an attacker can easily weaponize these design shortcomings by mounting a new class of devastating attacks, dubbed "cloak and dagger," which lead to the complete control of the UI feedback loop.
- We evaluate the practicality of these attacks with a user study. None of the 20 human subjects suspected anything, even after we revealed the malicious nature of the app they interacted with.
- We propose a defense mechanism that can block any attempt to confuse the end-user and limit the (malicious) capabilities of the accessibility service.

II. Two permissions

This section introduces relevant background information about the two permissions discussed in this work, including what capabilities they provide, how to enable them, and how they are used in real-world apps. The next section, instead, describes the existing security mechanisms that (attempt to) enforce that the powerful capabilities granted by these permissions cannot be abused.

A. The SYSTEM_ALERT_WINDOW permission

An app having the SYSTEM_ALERT_WINDOW permission has the capability to draw arbitrary overlays on top of every other app. According to the official documentation, "Very few apps should use this permission; these windows are intended for system-level interaction with the user." [6]. Despite this warning, the SYSTEM_ALERT_WINDOW is used by very popular apps such as Facebook, LastPass, Twitter, and Skype. In particular, we found that about 10.2% (454 out of 4,455) of top apps on Google Play Store require this permission. The most common usage of this permission is floating widgets, such as the ones implemented by music players, weather notification apps, and Facebook Messenger. Security apps such as app lockers, desk launchers, and password managers also use this permission to implement some of their key features.

It is important to mention that, starting from Android 6.0, this permission is treated differently from the others (such as the more traditional location-related permission). In particular, in the general case, the user needs to manually enable this permission through a dedicated menu. Thus, the general belief is that it is quite challenging for an app to obtain this permission. However, we observed that if an app is installed through the latest version of the official Play Store app, *the SYSTEM_ALERT_WINDOW permission is automatically granted*. Moreover, if the app targets an SDK API higher or equal than 23 (an app's developer can freely select which API level to support through the app's manifest), the Android framework will not show the list of required permissions at installation time: in fact, modern versions of Android ask the user to grant permissions at run-time. This means that, since the SYSTEM_ALERT_WINDOW permission is automatically granted, the user will not be notified at any point. We note that this behavior seems to appear a deliberate decision by Google, and not an oversight. To the best of our understanding, Google's rationale behind this decision is that an explicit security prompt would interfere too much with the user experience, especially because it is requested by apps used by hundreds of millions of users.

On the technical side, the overlay's behavior is controlled by a series of flags. The Android framework defines a very high number of flags, the three most important being the following:

- `FLAG_NOT_FOCUSABLE`: if set, the overlay will not get focus, so the user cannot send key or button events to it, which means the UI events sent to the overlay will go *through* it and will be received by the window behind it.
- `FLAG_NOT_TOUCH_MODAL`: if set, pointer events outside of the overlay will be sent to the window behind it. Otherwise, the overlay will consume all pointer events, no matter whether they are inside of the overlay or not. Note that setting the previous flag implicitly sets this one as well.
- `FLAG_WATCH_OUTSIDE_TOUCH`: if set, the overlay can receive a single special `MotionEvent` with the action `MotionEvent.ACTION_OUTSIDE` for touches that occur outside of its area.

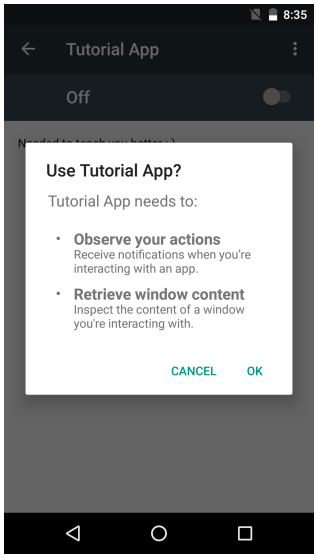
If none of the above flags are specified, the overlay will receive all events related to the user interaction. However, in this case, these events cannot be propagated to the window *below* the overlay. There are several other flags and technical aspects that are relevant for our work: for clarity reasons, we postpone their discussion to later in the paper.

B. The Accessibility Service

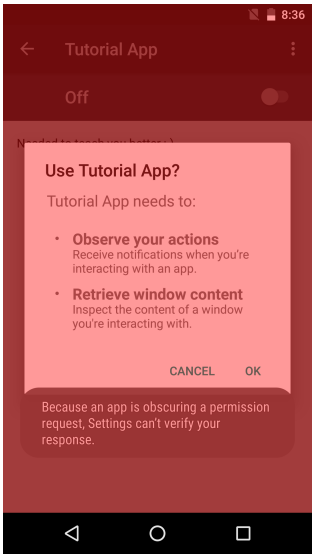
The accessibility service is a mechanism that is designed to allow Android apps to assist users with disabilities. In particular, an app with this permission has several powerful capabilities. In fact, the app is notified (through a callback-based mechanism) of any event that affects the device. For example, the main `onAccessibilityEvent()` callback is invoked whenever the user clicks on a widget, whenever there is a "focus change," or even when a notification is displayed. The details about these events are stored in an `AccessibilityEvent` object, which contains the package name of the app that generated the event, the resource ID of the widget, and any text-based content stored by these widgets (note that the content of passwords-related widgets is not made accessible through this mechanism). This object thus contains enough information to reconstruct the full *context* during which the event has been generated.

Moreover, an accessibility service app can also access the full *view tree* and it can arbitrarily access any of the widgets in such tree, independently from which widget generated the initial accessibility event. An accessibility service can also perform so-called *actions*: for example, it can programmatically perform a *click* or a *scroll* action on any widget that supports such operations. It can also perform "global" actions, such as clicking on the back, the home, and the "recent" button of the navigation bar.

Google is aware of the security implications of this mechanism. Thus, the user needs to manually enable this permission through a dedicated menu in the Settings app.



(a) This figure shows the popup that informs the user about the security implications of enabling the accessibility service. To grant authorization, the user needs to click on the OK button.



(b) This figure shows the warning message that is shown when the user clicks on the OK button that is covered by an overlay. The overlay in the figure is drawn semi-transparent for clarity. Of course, during a real attack, the overlay would be drawn completely opaque.

Fig. 1: Accessibility service popup & security mechanism

Not surprisingly, this permission is less popular than the SYSTEM_ALERT_WINDOW permission: Among the top 4,455 apps on the Play Store, we find 24 apps that use the accessibility service. It is worth noting that *none* of them are purely designed for people with disabilities. In fact, most of them are security apps such as password managers (e.g., LastPass), app lockers, desk launchers, and antivirus apps. We also found that 17 of these apps require both permissions discussed in this paper. Several of these apps are installed and used by more than one hundred million of users.

III. Existing Security Mechanisms

The two permissions discussed thus far grant capabilities that are clearly security-related. However, although powerful, the actions that an app can perform must adhere to what specified in the documentation and to what is communicated to the user. Thus, the Android OS implements a number of security mechanisms to prevent abuse.

Security Mechanism #1. The SYSTEM_ALERT_WINDOW permission allows an app to create custom views and widgets on top of any another app. However, the system is designed so that the following constraint always holds: if an overlay is marked as "pass through" (that is, it will not *capture* clicks), the app that created the overlay will not know when the user clicks on it (however, since the overlay is pass through, the click will possibly reach what is below); instead, if the overlay is created to be "clickable," the app that created it will be notified when a click occurs, and it will also have access to its exact coordinates. However, the system is designed so that an overlay cannot propagate the click to the underlying app. This is a very fundamental security mechanism: in fact, if an app could create an (invisible) overlay so that it could intercept the click *and* also propagate it to the app below, it would be trivial, for example, to record all user's keystrokes: a malicious app could create several overlays on top of all keyboard's button and monitor user's actions. At the same time, since the overlays are invisible and since the clicks would reach the underlying keyboard, the user would not suspect anything.

An overlay can also be created with the `FLAG_WATCH_OUTSIDE_TOUCH` flag, such that the overlay will be notified of any clicks, even if they fall outside the app itself. However, once again for security reasons, the event's precise coordinates are set only if the click lands in the originating app, while they are set to `(0,0)` if the click lands on a different app. In this way, the attacker cannot infer *where* the user clicked by using the coordinates. This mechanism makes also difficult to mount practical clickjacking attacks: in fact, it prevents an app to lure the user to click on what's below and at the same time being notified exactly where the user clicked, and it is thus not trivial to infer whether the user has been successfully fooled. This, in turn, makes mounting multi-stage UI redress attacks challenging, since there is currently no reliable technique to know when to advance to the *next stage*.

Security Mechanism #2. An accessibility service app has access, by design, to the content displayed on the screen by the apps the user is interacting with. Although the accessibility service does not have access to passwords (see below for a more detailed discussion), it does have privacy-related implications. Thus, in Android, the service needs to be manually enabled by the user: after pressing on the "enable" switch, the system shows to the user an informative popup (as in Figure 1a) and she needs to acknowledge it by pressing on the OK button.

Security Mechanism #3. Given the security implications of the accessibility service, the Android OS has a security mechanism in place that aims at guaranteeing that other apps cannot interfere during the approval process (i.e., when the user is clicking on the OK button). This defense has been introduced only recently, after a security researcher showed that it was possible to cover the OK button and the popup itself with an opaque, passthrough overlay: while the user is convinced to interact with the app-generated overlay, she is actually authorizing the accessibility service permission by unknowingly pressing OK [7].

The new security mechanism works in the following way. For each click, the receiving widget receives a `MotionEvent` object that stores the relevant information. Among these information, Google added the `FLAG_WINDOW_IS_OBSCURED` flag (*obscured* flag, in short). This flag is set to true if and only if the click event passed through a different overlay before reaching its final destination (e.g., the OK button). Thus, the receiving object (once again, the OK button in our case) can check whether

this flag is set or not, and it can decide to discard the click or to take additional precautions to confirm the user's intent. Figure 1b shows the message shown when the user clicks on the OK button while an overlay is drawn on top. We inspected the Android framework codebase and we found that this flag is used to protect the accessibility service, but also to protect the `Switch` widgets used to authorize each individual permission. Google is advising third-party developers to use a similar approach to protect security-sensitive applications.

Security Mechanism #4. To maximize the usefulness of accessibility service apps, they are given access to the content displayed on the screen. However, for security reasons, they are not given access to highly private information, such as password. This is implemented by stripping out the content of `EditText` widgets known to contain passwords. [8]

IV. Attacking the UI Feedback Loop

As we have mentioned in the introduction, the ultimate strength of "cloak and dagger" attacks lies in their complete control of the UI feedback loop between what users see on the screen, what they input, and how the screen reacts to that input. From a more conceptual point of view, the UI offers an I/O channel to communicate with the user. In turn, the two directions of the channel can be attacked in an active or a passive fashion. This leads to four distinct attack primitives, which we discuss next.

Primitive #1: Modify What The User Sees. An attacker may want to confuse or mislead the user by showing her something other than what is displayed on the screen. For example, in the context of clickjacking, the attacker may want to modify the prompt displayed by the system to trick the user into clicking "yes." In other scenarios, instead, the attacker may want to hijack the user's attention, for example by launching an attack while the user is distracted watching a video.

Primitive #2: Know What is Currently Displayed. Before we can properly modify what the user sees, we need to know what we are modifying. Continuing with the clickjacking example, our attack can only be successful if we know the system is displaying the targeted prompt: if we show our modified prompt when the target is not even on the screen, we will alert the user that something is wrong. As another example, to steal the user's password with a fake Facebook login, it only makes sense to show the fake UI when the real one is expected. In general, an attacker aims at determining which app is on top and which activity is displayed at any given time.

Primitive #3: User Input Injection. This primitive allows an attacker to control the user's device, while all the previous primitives provide proper "masking" for the effect of user input injection. In particular, to disable specific security features or to silently install an additional app, the attacker needs, for example, to inject clicks into the Android Settings app.

Primitive #4: Know What the User Inputs (and When). The attacker may want to monitor relevant *GUI events*, such as user clicks. In some cases, this is necessary so the attacker can update the modified display in Primitive #1 to maintain the expected user experience. With the clickjacking example, it is necessary to know that the user has clicked on either "yes" or "no" to dismiss the fake message we are displaying. This primitive can also be used to leak the user's private information.

V. Design Shortcomings

We identified four different design choices/shortcomings in Android that either enable easy implementation of the attack primitives, or make it harder to defend against cloak and dagger attacks.

Design Shortcoming #1. The main capability granted by the SYSTEM_ALERT_WINDOW permission is the ability to draw windows on top of other windows. In particular, *an app can draw arbitrary windows (in terms of their shape, appearance, and position) at arbitrary times*. This provides a first step to implement Attack Primitive #1.

Design Shortcoming #2. Regarding the accessibility service: *all GUI objects on the screen are by default treated equal*. As such, any app that has the BIND_ACCESSIBILITY_SERVICE permission can easily implement both Attack Primitives #2 and #4 to receive necessary information from most apps, and implement Attack Primitive #3 to inject input to many apps. The only exceptions are apps that declare some of their widgets as security-sensitive or choose to override the default behavior of how their widgets provide information to the accessibility service. The most prominent example of the former exception is the `getText()` method of password-related `EditText` widgets that returns an empty string (instead of returning its content). Another security consequence of this default "on" policy towards the accessibility service is that it is very easy to overlook apps/widgets that need to override the default for security reasons, as we will demonstrate in Section VI.

Design Shortcoming #3. As just mentioned, in Android an app can create a number of windows that are all completely customizable. The customization is related to the look and feel of the window, and provides what is necessary to implement Attack Primitive #1 in a completely stealthy manner (i.e., any overlay can look like part of the UI it's overlaying without any visible difference). Furthermore, it is also possible to define callbacks (mostly related to the graphical UI) and the window's behavior when clicked. Moreover, these callbacks receive specific objects (e.g., `MotionEvent`) that provide several information about the context. We show that *the inherent complexity of the `WindowManager` leads to the creation of unexpected side channels*, and possibly provides an alternative method to implement Attack Primitives #2 and #4.

Design Shortcoming #4. By design, *Android apps do not have access to the current context*. For example, an app cannot know whether, at a given point in time, there is another app displayed on top of it. While this does not necessarily help the implementation of any of our Attack Primitives, it certainly makes it very hard for individual apps to defend against cloak

and dagger attacks. To the best of our knowledge, the only Android feature that provides some useful information for apps to know their UI is being attacked is Security Mechanism #3, but we will show that this mechanism is not always effective. The key observation in this design shortcoming is that an app does not have any capability to determine whether it should trust the user input and it does not know whether the user had access to enough information to take an informed decision.

We note there is an interesting trade-off between this design shortcoming and Design Shortcoming #3 (DS#3): the more contextual information the Android framework exposes to an Android app, the more information a malicious app has access to implement attacks. We also note that while Design Shortcoming #2 may have security-related repercussions, it does save developers a lot of efforts in making their apps accessible to people with disabilities.

VI. Unleashing Mayhem

This section discusses how an attacker can weaponize the design shortcomings discussed thus far. Moreover, we show how, in some cases, the existing security mechanisms themselves can be used as an attack vector. All the attacks discussed in this section have been tested on a Nexus 5 running Android 6.0.1, and they are still unaddressed at the time of writing. Table I in Appendix A systematizes the main aspects of these attacks.

A. Clickjacking Made Practical

Attack #1: Context-aware Clickjacking. One known attack in Android is the possibility to perform clickjacking attacks. These attacks work by spawning a security-sensitive app, which we call the *target app*, and by creating an on-top, opaque overlay that does not capture any of the user interaction: while the user believes she is interacting with the app she sees, she is in fact interacting with the target app in the background. In a security context, the target app would usually be the Android Settings app, or any other "privileged" app. The malware would then use social engineering techniques to lure the user to click on specific points on the screen.

Clickjacking is relevant to our work because, very recently, a security researcher discovered that it is possible to perform clickjacking to lure the user to unknowingly enable the accessibility service [7]. In response to the researcher's report, Google implemented the security mechanism based on the `FLAG_WINDOW_IS_OBSCURED` flag described in Section III as *Security Mechanism #3*. The researcher subsequently discovered that the current implementation of this defense mechanism only checks that the portion of the clicked button was not covered by any overlay, and it does not guarantee that the OK button is visible in its entirety. According to Google, this issue does not pose any concrete and practical risks, and, reportedly, there are no plans to fix it.

The main limitation of current clickjacking techniques is that the malicious app does *not* know when and where the user clicked (this is ensured by Security Mechanism #1 described in Section III) and so it does not have precise control on when to move to the next step, making the attack less practical.

We developed a technique that makes clickjacking aware of the user's actions. We call this new technique *context-aware clickjacking*. Our technique works by creating a full screen, opaque overlay that catches *all* user's clicks, except for the clicks in a very specific position on the screen (the point where the attacker wants the user to click). This is achieved by actually creating multiple overlays so to form a *hole*: the overlays around the hole would capture all the clicks, while the overlay on the hole would make the clicks pass through. The key observation that makes this technique context-aware is the following: since there is only *one* hole, there is only one way for a user's click to *not* reach the malicious app. This observation makes it possible to use the *Security Mechanism #1* as a side channel: if the event's coordinates are set to `(0,0)`, then it means that the user clicked exactly in the hole (otherwise, the event's coordinates would be set to their actual value).

This feature makes these attacks particularly practical. For example, when performing clickjacking to lure the user into enabling the accessibility service, the attacker just needs to hijack three clicks: these three clicks do *not* need to be consecutive or near in time. Moreover, the malicious app knows exactly when the user clicked on the hole. This gives the app enough information to know when to move to the *next step*, and to update the overlay shown to the user so to appropriately *react* to the click, even if the click never reached the malicious app.

Attack #2: Context Hiding. In Section III we described how the Android OS features a security mechanism based on the *obscured* flag: an object receiving a click event can check whether the user click "passed through" an overlay. We also mentioned that a security researcher determined how this mechanism is implemented in an insecure way: as long as the user clicks on a part that is not covered, the flag is not set.

We argue that this defense mechanism would be insecure even if it were implemented correctly. In fact, we believe this mechanism is vulnerable *by design*: if the user can only see the OK button, how can she know what she is authorizing? Is she clicking the OK button on an innocuous game-related popup, or is she unknowingly enabling the accessibility service? By using context-aware clickjacking, it is easy to create overlays with a single hole in correspondence to the OK button, thus completely covering all security-relevant information. Thus, a malicious app can hide the real context from the user, and it can lure her into clicking on the OK button – *even if the OK button is entirely visible*.

As we will discuss in Section VII, we performed a user study that evaluated how practical it is to lure users to enable the accessibility service even if the obscured flag implementation were correct: *none* of the human subjects involved in our user study suspected they were under attack.

B. Keystroke Recording

We now describe three new attack vectors to record all user's keystrokes, including sensitive information such as passwords. The first attack only relies on the

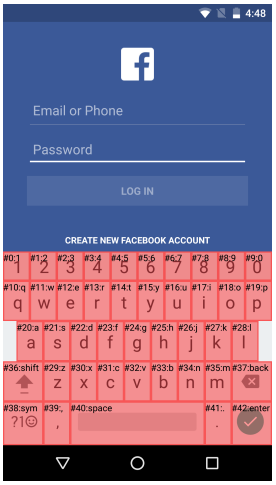


Fig. 2: This figure shows the organization of the overlays created for our "Keystroke Inference" attack (Attack #3). Of course, the overlays are made visible only to ease our explanation: the overlays would be created invisible during an actual attack.

SYSTEM_ALERT_WINDOW permission and exploits DS#1 and DS#3, the second attack only relies on the BIND_ACCESSIBILITY_SERVICE and exploits DS#2, while the third attack relies on the combination of the two.

Attack #3: Keystroke Inference. This attack is based on a novel technique that attempts to circumvent *Security Mechanism #1*. In particular, we show how it is possible to use the well-intentioned obscured flag recently introduced by Google as a side channel to infer where the user clicked. The net result of this attack is that an app with just the SYSTEM_ALERT_WINDOW permission can record all keystrokes from the user, including private messages and passwords.

The attack works in several steps. We first create several small overlays, one on top of each key on the keyboard, as shown in Figure 2. Of course, during a real attack, these overlays would be completely transparent and thus invisible for the user. The overlays are created with the following flags: `TYPE_SYSTEM_ALERT`, `FLAG_NOT_FOCUSABLE`, `FLAG_NOT_TOUCHABLE`, and `FLAG_WATCH_OUTSIDE_TOUCH`. These flags make sure that each overlay does not intercept any click by the user (that is, when the user clicks on a keyboard's key, the click will reach the keyboard, as the user would expect). However, note that, thanks to the `FLAG_WATCH_OUTSIDE_TOUCH`, *each* overlay receives a `MotionEvent` object for each click. As described in Section III, these click events do not contain any information about where the user actually clicked.

However, we discovered that it is possible to use Google's obscured flag as a side channel to infer where the user actually clicked. This attack works thanks to the following two observations. First, the overlays are created in a very specific, attacker-known order, and they are organized as in a stack: overlay #0 (top left in Figure 2) is at the bottom of the stack, while overlay #42 (bottom right) is at the top: thus, each of these overlays has a different Z-level. The second observation is that, for each overlay, the obscured flag is set depending on whether or not the user clicked on an overlay that was on top of it. For example, if the user clicks on overlay #0, the `MotionEvent` events received by each overlay will have the obscured flag set to 0. However, if the user clicks on overlay #1, the event delivered to overlay #0 will have the obscured flag set to 1. More in general, we observed that if the user clicks on overlay #i, the events delivered to overlays #0 $\rightarrow$ #(i-1) will have the obscured flag set to 1, while all the events delivered to the remaining overlays will have the flag set to 0.

Thus, by creating the overlays in a specific order and by collecting the events and their obscured flags, it is effectively possible to use the obscured flags as a side channel to determine on which overlay the user has clicked, thus breaking *Security Mechanism #1*. We were able to write a proof-of-concept that can infer all keystrokes in a completely deterministic manner.

Attack #4: Keyboard App Hijacking. The accessibility service is a powerful mechanism that has access to the content displayed by the apps the user is interacting with. However, for security reasons, it is designed so that it cannot get access to security sensitive information, such as passwords. In practice, this is implemented so that when an app attempts to read the content of an `EditText` widget containing a password, the `getText()` method always returns an empty string. This behavior is documented at [8]: "[...] any event fired in response to manipulation of a PASSWORD field does NOT CONTAIN the text of the password."

It is possible to use DS#2 to bypass this protection mechanism. In fact, by specifying the `FLAG_RETRIEVE_INTERACTIVE_WINDOWS` (according to the documentation, it indicates to the system that "the accessibility service wants to access content of all interactive windows" [9]), the keyboard app itself (package name: `com.google.android.inputmethod.latin`) is treated as a normal, unprivileged app, and each of the *key* widget generates accessibility events through which it is easy to record *all* user's keystrokes, including passwords.

Attack #5: Password Stealing. Attacks #3 and #4 show that it is possible to abuse DS#1, DS#2, and DS#3 to record all user's keystrokes.

Here we describe an additional attack that uses a combination of the two. The attack works in several steps. First, the attacker uses the accessibility service to detect that, for example, the user just opened the Facebook app's login activity. At this point, the malicious app uses the overlay permission to draw an overlay that looks like the username and password `EditText` widgets. Note how, differently from the previous attacks, the widgets are actually visible: however, they match exactly the Facebook user interface, and the user does not have any chance to notice them. Thus, the unsuspecting user will interact with the malicious overlays and will type her credentials right into the malicious app.

To make the attack unnoticeable to the user, the malicious app would also create an overlay on top of the login button: when the user clicks on it, the malicious overlay would catch the click, fill in the real username and password widget in the Facebook app, and finally click on the real Facebook's login button. At the end of the attack, the user is logged in her *real* Facebook account, leaving her completely unsuspecting. Moreover, by checking whether the login attempt was successful, our attack can also confirm that the just-inserted credentials were correct.

We note that this technique is generic and it can be applied to attack any app (e.g., Bank of America app). Moreover, the malicious app would not need to contain any code of the legitimate app, thus making it more challenging to be detected by repackaging detection approaches. Our attack, in fact, replaces many of the use cases of repackaging-based attacks, making them superfluous.

C. Unlocking The Device

We now describe two attacks related to the Android locking mechanisms. These two attacks are possible due to DS#2.

Attack #6: Security PIN Stealing. We discovered that the security screen pad used by the user to digit her PIN to unlock the device generates accessibility events, and that an accessibility app is able to receive and process the events even when the phone is locked. In fact, we discovered that the `Button` widgets composing the security pad are treated as normal, unprotected buttons and that their content description contains the number or symbol represented by the button. Thus, any app with accessibility service can easily infer which buttons the user is pressing and can thus infer which is the user's PIN. We believe the user's PIN should be considered as sensitive as user's passwords: first, it is possible that the user reuses the PIN in other settings; second, and more importantly, the attacker armed with the knowledge of the PIN can use it to unlock the phone in case of physical access to the device, or it can use it to perform other attacks, as the one we describe next.

Attack #7: Phone Screen Unlocking. We discovered that apps with accessibility service not only can receive events while the phone is locked, but they can also inject events. We discovered it is possible for an accessibility app to unlock the device in case a "secure lock screen" is not used. We also discovered an app can inject events onto the security pad and it can click and "digit" the security PIN (which can be inferred by using the previous attack). To make things worse, we noticed that the accessibility app can inject the events, unlock the phone, and interact with any other app *while the phone screen remains off*. That is, an attacker can perform a series of malicious operations with the screen completely off and, at the end, it can lock the phone back, leaving the user completely in the dark.

We note that this attack works if the user did not setup a security lock, or if she setup a security PIN. However, this attack does not work if the user setup a secure pattern: to the best of our knowledge, it is not possible to inject "swipe" events through accessibility service. Interestingly, at a first impression it appears that to unlock the phone (even in case of security PIN) one would need to "swipe up" the security lock icon. However, it turns out that by injecting a "click event" on the lock icon, the security pad appears and the accessibility app can inject the PIN and unlock the phone. We also note that, according to the documentation, not even an app that enjoys full admin privileges should have the possibility to unlock the device.

The fact that an app can unlock the device and stealthily perform actions could be also combined with other attacks that generate revenue: in fact, it is simple to imagine a malware that would unlock the phone during the night, and it would go on the web and click ads (a scenario we discuss later in this section) or perform any other actions that directly generate revenue.

D. From Two Permissions to God Mode

Given an app with the SYSTEM_ALERT_WINDOW and the BIND_ACCESSIBILITY_SERVICE permissions, we show how it is possible to install a second malicious app that requires *all* permissions and how to enable all of them, while keeping the user unsuspecting.

Attack #8: Silent App Installation. The initial malicious app (the one with only the two permissions discussed in this paper) can embed a secondary, much more malicious app in the form of an APK. Thus, an app can initiate the app installation by sending an `Intent` with `ACTION_VIEW` as action, and `Uri.parse("file:///<path>/malware.apk")`, `"application/vnd.android.package-archive"` as additional data. This will generate a prompt asking the user for confirmation, at which point the app can automatically click on the "Install" button. Before doing that, however, the app checks whether side-loading of third-party app is enabled: if it is not enabled, the app opens the Settings app, browses to the security settings, and automatically grants itself permission to side-load additional apps.

Since the "Install" button is unprotected, it is possible to perform this attack while stealthily covering the screen with an on-top overlay. In fact, as we discuss later in the paper, we did test the practicality of this approach by performing this (and the following) attack while the user believed to be watching an innocuous video. The app can then cover its track by opening the "recent windows" and by dismissing them all.

Attack #9: Enabling All Permissions. Once the secondary malicious app is installed, it is trivial for the initial app to grant device admin privileges: we found that the "Enable" button is unprotected and it can be easily clicked through an accessibility event. However, the `Switch` widgets that the user needs to click are protected by the `FLAG_WINDOW_IS_OBSCURED` flag. It turns out that the current implementation of the mechanism that sets and propagates this security flag handles user-generated clicks and clicks generated by an accessibility service app in a different way. In particular, we observed that the flag is *not* set for any event generated by the accessibility service. Thus, we found

it is possible to automatically click and enable all permissions while a full screen overlay is on top. At the end of this and the previous attack, the initial unwanted application was able to silently install a God-mode malicious app.

E. Beyond the Phone

Attack #10: 2FA Token Stealing. We show that a malicious app that has access to the accessibility service is able to steal two-factor authentication codes stored on or received by the phone. In fact, for SMS-based tokens, the accessibility service can be configured to be notified for any new notification, which contains the required token. For tokens of other natures, such as the ones offered by the Google Authenticator app [10], the malicious app can easily open the activity displaying the tokens and easily read them off the screen: in fact, we found that none of the views containing the tokens are protected. To make things worse, this attack can be combined with the fact that the phone can be unlocked and that the malware can perform these operations while the screen is off. The malware can also generate a token "when needed," and then get rid of the notification.

Attack #11: Ad Hijacking. An accessibility service app is notified of all GUI-related event that happens. These events include "click" and "touch" events, but also "windows change" events when a graphical interface is redrawn. The app also gets access to the app (identified by its package name) that generates these events. The app would also get access to the entire view tree, which includes details such as the type of each widget in the tree, their size, and their position on the screen. This information can be used by a malicious app to fingerprint each activity of each app and identify where and when ads are shown. At this point, the malicious app can draw an invisible on-top overlay on top of the ad: whenever the user would press the ad, the malicious app would intercept the click and redirect the user to the malware author-owned ad, which would generate revenue.

Industry researchers recently discovered malware samples that abuse the accessibility service to automatically install adware apps and to automatically click on ads generated by them. While this is profitable, the user would be clearly aware of what is happening, and it would attempt to uninstall the malicious apps or, more likely, to factory reset her phone. We note that while these malware samples also abuse accessibility service, their malicious process is very different. In particular, our attack is completely stealthy, and none of the parties involved (i.e., the user, the OS, the app developer) has any chance to notice the fraud.

Attack #12: Exploring the Web. Among other things, we discovered that the accessibility service has full access to the phone's browser. It is easy to write a program to open Chrome and visit an arbitrary page. To make things worse, the content of the page is automatically parsed by accessibility service and easy-to-use Android objects are provided for each component shown on the target HTML page. For example, the HTML page itself is made accessible through a View tree and an HTML button is converted to a Button Android widget. The accessibility service not only does it have access to the information, but it is also able to interact with it by, for example, performing "click" actions. Once again, it is simple to imagine malware that would secretly unlock the phone and click on "like" buttons on social networks and post content on behalf of the user. Moreover, since all HTML elements are nicely exposed, we believe it would be simple to extend the "password stealer" attacks we described earlier to web-based forms.

F. Putting All Together

All the attacks that we have described start off by just luring an user to install a single app hosted on the Play Store. This app's manifest would declare only two permissions: the SYSTEM_ALERT_WINDOW and the BIND_ACCESSIBILITY_SERVICE permissions. A quick experiment shows that it is trivial to get such an app accepted on the Play Store. In particular, we submitted an app requiring these two permissions and containing a non-obfuscated functionality to download and execute arbitrary code, and this app got approved after just a few hours. Since this app targets a recent Android SDK, at installation time the user would not see any prompt related to the required permission, thus leaving him completely unsuspecting. Moreover, since the SYSTEM_ALERT_WINDOW permission is automatically granted for apps hosted on the Play Store, the only missing step for the malicious app is to lure the user to enable the BIND_ACCESSIBILITY_SERVICE permission.

However, as we described earlier, it is possible to perform clickjacking attacks to lure the user into enabling the BIND_ACCESSIBILITY_SERVICE permission: thanks to our new context-aware clickjacking attack, this can be done in a very practical and deterministic manner. Our user study, described in Section VII shows that *none* of the human subjects involved suspected anything. In fact, the attacker just needs to hijack three clicks from the user, and since the app has full control of the context, these clicks do not even need to be sequential. After the user's three clicks, the device is fully compromised and, to make things worse, the user does not even have a chance to suspect anything is wrong.

At this point, the malicious app can enable side-loading, install another app (downloaded at run-time), enable all its permissions, including device admin and accessibility service, and launch it: a God-mode app is now installed on the device. Given the attacks we described, the malicious app would now be able to wait until night, to then silently unlock the phone and perform the attacks we described earlier, including leaking all user's credentials (including passwords, security PINs, and two-factor authentication codes), ad jacking, and browsing the web and leaking even more data. Since the app has all permissions enabled, the malware could perform the variety of malicious actions described in the literature (e.g., record audio and video, steal user's contacts), the imagination being the only limit.

We note that the initial malicious app has even the chance to clean after its steps. For example, the app could disable the side-loading option. As another example, consider a

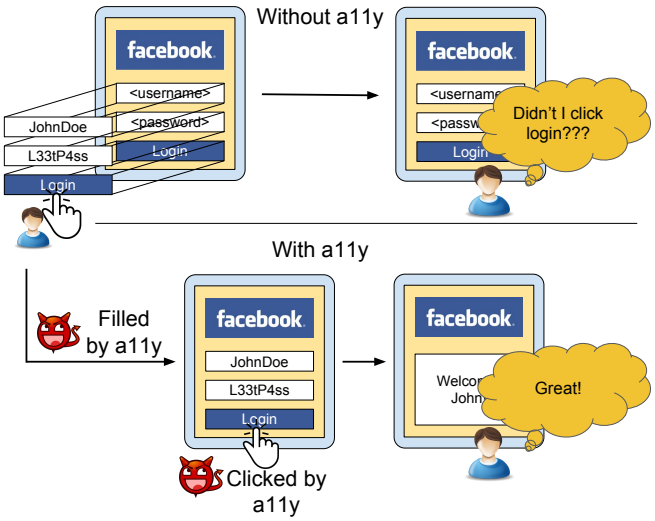


Fig. 4: This figure shows how our attack stealthily steals the user's credentials by using a fake UI and completing the login with the help of accessibility service vs. using only a fake UI (e.g., [11], [12], [1], [2], [3]).

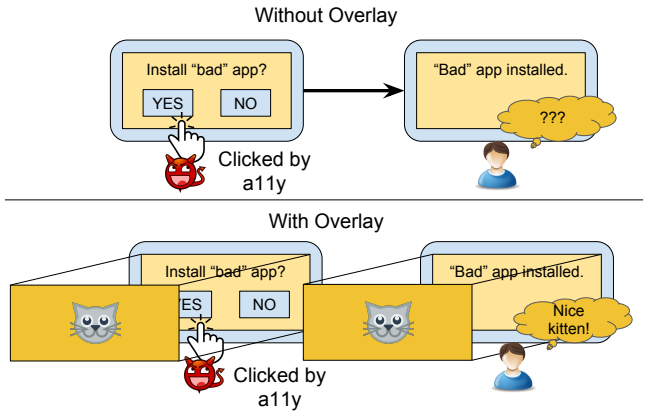


Fig. 3: This figure shows how our attack stealthily installs a malicious app using the accessibility service while covering its action using an overlay vs. using only accessibility service (e.g., [4]).

scenario where the initial app is installed through social engineering, by making the user believe that this app is, for example, a Pokemon Go-related app: to delete all traces, once the secondary malicious app has been installed, the initial malicious app could silently install the *real* Pokemon Go app from the Play Store, and it could then uninstall itself: at this point, the user would believe she has interacted with the legitimate app from the very beginning.

To make things worse, we finally note that the new app has a series of ideal properties for malware. In fact, the secondary installed app does not need to be hosted on the Play Store, but it can be dynamically downloaded: thus, the attacker has full flexibility when generating this new app, and it could even use polymorphism to make it very challenging to be tracked. It is also possible to configure the malicious app to not appear in the app's launcher (this is possible by removing `android.intent.category.LAUNCHER` from the app's manifest). Last, since the malicious app is granted with device admin privileges, its uninstall button will be disabled, and it is thus easier to be disguised as a legitimate system app. Of course, it is possible that a determined user will be able to spot this malicious app. However, since the attacks described in this paper are designed to be stealthy, the user would not even think her device is compromised.

VII. User Study

Some of the attacks we presented require interaction with the end user. We designed and performed a user study to evaluate the practicality of these attacks. In particular, we evaluated the practicality of the attacks related to clickjacking (#1 & #2), silent installation of God-mode app (attacks #8 & #9), and the most complex of the keystroke recording attack (attack #5). We start this section with a description of the user study, and then discuss the results. Our study was reviewed and approved by our institution's internal review board (IRB).

A. Experiment Description

Human Subjects Recruitment. For our study, we were able to recruit 20 human subjects. We performed the recruitment by advertising our study through mailing lists and word-of-mouth. The subjects involved in our study have a variety of different backgrounds, ranging from doctoral students, post-doctoral researchers, and personnel involved in the administration. All the participants were recruited at the research institution where the study has been conducted. The only requirement to participate to the study was to have at least a minimum familiarity with Android devices. While recruiting the participants, we ensured that they were not aware of any details related to our experiment.

Experiment Settings. Before starting the experiment, we provided a bit of context to the human subject. However, since our main goal is to determine the stealthiness of our attacks, we obviously could not state our actual intent. Thus, we created a fictitious scenario in which we, the researchers, created a novel security mechanism for Android and that we wanted to test whether this new mechanism would introduce noticeable effects.[3] As we discuss throughout this section, the user is first asked to interact with an app we wrote. This app only requires the two permissions we focus on in this paper. Moreover, we simulate a scenario in which the user downloaded the app from the official Play Store: thus, the SYSTEM_ALERT_WINDOW permission is already granted, while the BIND_ACCESSIBILITY_SERVICE permission is not. The user will be lured to enable this second permission as the user study progresses.

[3]Given the nature of our study, we needed to use concealment. We note that we properly included the description of this element while preparing our application package to obtain IRB approval. We also note that, as per IRB guidelines, at the end of the experiment each of the subject was debriefed about the real nature of the study.

Experiment Organization. The study is organized as follows. First, we ask the user to answer few preliminary questions. The main goal of these initial questions is to assess whether the subject had familiarity or owned an Android device. After we have established that the subject satisfies our inclusion criteria, we proceed with the main part of our study, which is a controlled experiment designed to test whether the user can tell whether they are under attack or not. We did this for two sets of attacks in Section VI, namely attacks #1, #2, #8, #9 (for enabling accessibility and installing the God-mode app) and attack #5 (for stealing Facebook password). In particular, we ask the user to interact with an app we wrote (running on a device we provide) twice for each set of tested attacks, one where the attack actually happened and one "control run" where the attack was not launched. We randomized the order of the runs.[4] After the experiment for each of the tested set of attacks, we asked the subject few questions to assess whether she had noticed anything during the two runs.

As a final step of our study, we give the subject the possibility to freely interact with the device (which now has accessibility enabled for our app, and the God-mode app installed) and we ask them to report any "weirdness" the subject would like to note. We then ask a series of wrap-up questions to assess to which extent the subject realized what actually happened during the experiment. The remainder of this section provides the details about the two phases, the assessment and wrap-up questions, as well as the results of the study.

Phase #1: Clickjacking and Silent App Installation. This phase aims at testing the practicality of the attacks #1 and #2 (related to clickjacking), and #8 and #9 (related to the silent installation of a God-mode app). For this phase, we ask the subject to interact with one app we wrote. We pre-installed this app on our test device before the beginning of the experiment. This app is disguised as a simple tutorial. The user is asked to click on three buttons: "Start Tutorial," "Next," and "OK." After the subject clicks on the final OK button, the app shows a short animation (for a duration of about 20 seconds). Once the video is over, a final message is shown. This marks the end of the first phase.

Unbeknownst to the user, what actually happens during the attack run is that the first three clicks enable the accessibility service: the first one clicks on the entry related to our app, the second one clicks on the "on/off" `Switch` widget, while the final click on the OK button actually clicks on the *real* OK button; *i.e., we are showing them overlays with text related to our tutorial, with holes in the overlay to show buttons that belong to the Settings app below*. At this point, our malicious app uses the SYSTEM_ALERT_WINDOW permission to create an on-top overlay that shows the video and, in the meantime, it uses the BIND_ACCESSIBILITY_SERVICE permission to silently enable side-loading of third-party apps and install a secondary, much more malicious app (which was embedded in the first). Then, it enables all its permissions, grants it device admin capabilities, and before the video is over, dismisses all the "recent" views so that the user will not notice anything once the attack is over. The secondary app that is installed is disguised as "Google Parental Control" app. Figure 3 depicts how our attack #8 can achieve its stealthiness, and it shows how our attack improves over existing ones.

As for the control run, we showed the user real UIs with the same text and buttons that are actually generated by our app, and, after the user clicks all three buttons, we show them the same tutorial video.

Phase #2: Password Stealer. This phase aims at evaluating the practicality of attack #5. In particular, we asked the subject to login on Facebook (with test credentials we provided). The subject needed to open the Facebook app, type the password in (the username was pre-filled by the app's cache), and click on the "Log In" button. As in the previous case, this phase is executed twice: in one of the two cases our malicious app performed the attack, while it remained idle in the other one.

In this case, during the attack run, our malicious app would use the accessibility service to infer that the user had opened the Facebook app, and it would then create an overlay that would resemble the password and login button widgets used by the real Facebook app (since the username is stored in a non-protected `EditText` widget, the malware can steal it by simply using the `getText()`'s accessibility service API). The unsuspecting subject would thus interact with our malicious widgets, type her password in, and click on the login button. At this point, our malicious app would copy the (now leaked) password to the real Facebook app, click on the real login button, and dismiss the malicious overlays. On the other hand, no overlay is created on top of the Facebook app during the "control run." Figure 4 graphically depicts this attack and it demonstrates its advantage over existing attacks that only use fake UI.

Assessment Questions. After each of the two phases, we asked several questions to assess whether the subject noticed any difference or any "weirdness" while performing the required steps. These questions include "Did you notice any difference?", "Did you notice any slowdown?", and "Did you notice any weirdness?". We allowed the subject to provide a yes/no answer as well as an open answer to let her clarify any of her thoughts.

Wrap-up Questions. Once the experiments for the two sets of attacks are concluded, we asked the subject to answer several questions in order to assess to which extent the subject understood what really happened during the experiments. We also took this opportunity to test the subject's knowledge on the two permissions we focus on in this paper. As the very last question, we reveal to the subject that the initial app was malicious: once again, we gave the subject free access to the device, and we ask her to note any impression, idea, or hypothesis about what actually happened during the experiment.

B. Results & Insights

We now describe the results of our user study. We organize the discussion by describing the results of each component of

[4] I.e., there is a 50% chance the first run will be the attack, and 50% the first run will be the control.

the attack: clickjacking, silent app installation, and password stealer. We then provide insights about the general awareness of the subjects.

Clickjacking. None of the 20 subjects noticed any weirdness while they were under attack. In particular, none of them reported noticing any difference between the two runs of the experiment. We conclude that our clickjacking attack is practical. We also note that this attack is practical even when assuming that the `FLAG_WINDOW_IS_OBSCURED` flag functionality were correctly implemented (that is, no overlay can obscure any part of the OK button).

Silent God-mode App Installation. Once again, none of the subject even suspected that they were under attack and they did not notice any significant difference between the two runs of the experiment. Interestingly, a few subjects did report some very general differences regarding some audio/video aspects of the video we showed. For example, two subjects reported that the image and sound quality decreased, or that the sound was higher in one case. One other subject reported that "the initial splash screen seems different." Lastly, two other subjects reported that the duration of the video changed between the two runs. However, we cannot find any correlations between the reported differences and whether a particular run is the attack run or not; to the contrary, for the reports about the video length, *one subject thought the video played while the attack was launched was shorter, while the other subject reported the exact opposite.* One hypothesis that explains why these subjects reported these differences is that they simply assumed there *was* a difference to be detected. As a matter of fact, quite a few participants smiled when asked "Did you notice any difference?": they then confessed they had no idea where the difference could have been.

Password Stealing. 18 out of the 20 human subjects did not detect any differences or weirdnesses between the two times they were asked to login into Facebook. The remaining two subjects triggered a bug in our implementation and, although they did not understand they were under attack, they did notice some graphical glitches. One of these two subjects miss-clicked on the widget holding the username and he noticed a graphical glitch. Nonetheless, the subject did not understand he was under attack, and he reported that he found differences in the "touch sensitiveness." The other subject encountered a different problem: instead of clicking on the "log in" button, he clicked on the "enter" keyboard's button, which our prototype did not handle correctly. Once again, the subject just noticed the glitch without realizing he was under attack. We note that these glitches are caused by simple imperfections of our prototype, and they are not caused by any conceptual limitation of our attacks.

At the end of the experiment, we explained to these two subjects the details about the attacks we were testing and why they encountered those graphical glitches. As an additional experiment, we asked them to repeat the experiment: Both subjects were *not* able to distinguish in which case they were under attack *even if they knew we were attacking them during one of the two runs*. We believe these results show that this attack is practical. We also note that this attack is the most complex (and most challenging to fix) among the other two other attacks to record the user's keystrokes (i.e., attacks #3 and #4).

Overall Awareness. None of the users was able to even suspect that they were under attack. What is more worrisome is that none of the users actually managed to understand what happened even after we told them the app they played with was malicious and even after we gave them complete and free access to the compromised device. We also stress that the subjects could not detect anything even when they had a chance to directly compare what happened against a "benign" baseline: In a real-world scenario, there would not be any baseline to use as reference. We also argue that it is quite easy to hide a malicious app among many benign ones. In fact, one subject opened the device admin settings and she found our seemingly benign "Google Parental Control" app: still, she did not suspect that app was malicious, and she assumed it was a system app. Only one subject noticed one aspect that was marginally related to our attack: the device was configured to enable the installation of side-loading app. While our attack turned this feature on, it is quite simple to "improve" the malware to reset the side-loading settings as it were before the attack started.

Finally, only two out of the 20 subjects knew what the SYSTEM_ALERT_WINDOW permission was and what an app with this permission can do. Instead, only five subjects knew what the accessibility service was (in fact, eight subjects in total declared to know about it, but three of them provided a wrong description). Interestingly enough, no single subject knew the details about both permissions.

We believe these results clearly indicate that the attacks we discuss in this paper are practical. Our results also indicate that the user's awareness of these permissions and thus of the risks they pose is currently quite low. We believe this is particularly worrisome because the user is not even notified about these two permissions and she will thus remain completely unsuspecting.

VIII. Discussion

The numerous attacks we described and the results of our user study highlight that the risks imposed by the SYSTEM_ALERT_WINDOW and BIND_ACCESSIBILITY_SERVICE permissions are currently vastly underestimated.

Responsible Disclosure. We responsibly disclosed all our findings to the Android security team, which promptly acknowledged the security issues we reported. When available, the fixes will be distributed through an over-the-air (OTA) update as part of the monthly Android Security Bulletins [13].

No Easy Fix. Unfortunately, even if we reported these problems several months ago, these vulnerabilities are still unpatched and potentially exploitable. This is due to the fact the majority of the presented attacks are possible due to the

inherent design issues outlined in Section V. Thus, it is challenging to develop and deploy security patches as they would require modifications of several core Android components. As an example, consider the issue related to *all widgets are treated equally*: to address this issue, the Android system would need to be modified so to add the concept of *type* of widget and so to react in different ways depending on such type. We argue we need a more principled design mechanism that directly focuses at addressing the design issues we uncovered or, alternatively, at making sure they cannot be weaponized to be a real concern.

IX. Securing The Android Graphical UI

This section proposes a series of modifications and enhancements to the Android system that directly address the design shortcomings we described in the previous section. Our proposed modifications are constituted by two main components: first, the introduction of *secure apps and widgets*, and, second, *system popups*. These new mechanisms would require system modifications, which we believe to be necessary. However, given the problematic state for which Android devices receive updates, we also discuss two recommendations that can be deployed within a much shorter timeframe.

A. Secure Apps & Widgets

We envision an extension of the Android framework that allows a developer to easily indicate to the OS that a given widget is security-sensitive. There are many typologies of widgets that a developer may want to flag as security-sensitive. Few examples are: `Button` or `Switch` widgets whose "click action" have a security-related semantics; `TextView` or `EditText` widgets that store passwords or two-factor authentication tokens; `Button` widgets for which merely knowing that they have been clicked might leak sensitive information (e.g., buttons on the keyboard app or on the security pad).

It should be as easy as possible for a developer to specify that a given widget is security-sensitive and, ideally, it should not require the implementation of any custom logic. We propose to define a new flag, which we call `FLAG_SECURITY_SENSITIVE` (or *secure flag*, in short), that a developer should be able to set for an arbitrary widget or view. As an additional convenient extension, the developer should be able to indicate to the OS that an entire activity, or even an entire app, should be considered as security-sensitive: in these cases, the secure flag should be conceptually propagated to all sub-views. The semantics of this flag is intuitive: it tells the Android OS that a given widget, activity, or app plays a security-related "role," and it thus needs special care.

This approach is preferable with respect to the current one because the developer does not need to implement any custom logic on how the app should "react" based on whether the `FLAG_WINDOW_IS_OBSCURED` flag is set. In fact, our proposal allows a developer to rely on a secure-by-design fallback such as "discard any interaction if the context is unsafe and notify the user." Moreover, the current approach only lets the developer detect problematic cases where an overlay is on top of a widget: as we have shown in this paper, this is only one of the many problems. We believe that a defense design system should be comprehensive. As a possible further extension, the developer would be given the possibility to customize how to "react" when an unsafe situation is encountered: however, *we believe that "custom logic" should be the exception, rather than the rule*.

Enforced Security Properties. We now define what are the properties that the OS should enforce for secure widgets (i.e., widgets for which the secure flag is set). First, whenever the user is interacting with an app that embeds any secure widget, no other app should be allowed to interfere with the user's interaction by being able to create arbitrary overlays on top. We note that this should apply not only for `Button` widgets, but also for `EditText` widgets that store user-entered passwords: When the user is interacting with "buttons" or "data" (of any kind) that are security-relevant, there should not be, by design, any possibility to (even subtly) interfere with it. (We note that there are few legitimate scenarios where this constraint might create practicality issues. We address these issues later in this section.)

Moreover, the OS should enforce that no input from the user should be accepted if there was something on top in the past few seconds. This precaution would make sure that the user has sufficient time and information to take informed decisions. This idea has been first proposed by Roesner et al. in [14].

We note that the proposed mechanism might cause backward compatibility issues: what would happen if a third-party app attempts to create a widget on top while the user is interacting with a secure app? One solution to address these issues is the following: instead of denying the possibility for an app to draw overlays, the OS would still allow it, but none of these widgets would be actually rendered while the user is interacting with a secure view. This would not break existing apps and it would ensure that the user cannot be misled.

Another property that should be enforced is that an accessibility service should not be able to automatically click or perform any action on any secure widgets: if the developer of an app would like to provide an option for an action to be performed programmatically, the developer should expose such functionality through an appropriate API (and protected by a permission, if needed). We note it is safe enough to let third-party accessibility apps to automatically *fill* `EditText` widgets, even if they store user-sensitive information. This would assure backward compatibility with popular apps such as LastPass.

Last, there should not be any direct or indirect mechanism to infer or leak information stored in secure widgets. As it already happens with password-related `EditText` widgets, the `getText()` method should return an empty string. The same principle should apply for `TextView` widgets that contain, for example, two factor authentication codes. We note that in some scenarios, the mere fact that a specific button or `TextView` has been clicked could also constitute a problem. For example, by knowing on which keyboard's `TextView` widget the user is clicking, it is trivial to record all keystrokes.

Thus, no accessibility events should be generated when the user interacts with these security-sensitive widgets.

Addressing Design Shortcomings. The Android API extension we just discussed directly addresses the design shortcomings we uncovered for this work. The new typology of widgets and apps, i.e., *secure widgets and apps*, addresses the shortcoming related to the fact that the Android OS treats all widgets equal (i.e., DS#2). Third-party apps are also now prevented to tamper with security-sensitive applications: while they are still able to create arbitrary widgets, our proposal effectively avoids the weaponization of such capability, thus effectively making DS#1 innocuous.

Our proposal also addresses DS#4: while apps would not have explicit control over the current context, they would have an OS-enforced guarantee that the context is safe enough to trust user's input. Interestingly, another option would have been to provide finer-grained information about the current context: which app is currently displaying widgets? In which positions? We argue this would be a bad decision. In fact, the more information is exposed to a third-party app, the more likely is that this information can be used as a side-channel to mount attacks (cf. DS#3). We believe our proposal hits the sweet spot in the trade-off between information provided to the app and possibility of having side channels.

Adoption. Once the modifications proposed in this paper are integrated within the Android OS, we envision Google leading the way and marking as "secure" critical apps such as the Android Settings app and the Keyboard app. Moreover, any widget that prompts the user for any security-relevant question should be marked as "secure" as well. Moreover, we believe developers of third-party apps would smoothly transition to using the proposed API extension: in fact, taking advantage of this system would require the usage of a single flag and, differently from the existing system, it would not require the implementation of any custom logic.

B. System Popups

Our proposal would be effective in addressing (or preventing abuse of) the design shortcomings we highlighted. However, our proposal has one side-effect that could cause compatibility problems: no app would be allowed to create any overlay when the user is about to input her credentials in a login activity (which should be assumed to be marked as "secure"); However, this mechanism might break certain legitimate functionalities. For example, when the current version of the LastPass app detects that an user is about to type her credentials in a login form, it creates an on-top overlay offering the user the possibility of using LastPass' auto-fill functionality.

We argue that what distinguishes this benign functionality and the attacks we presented is that, in this case, it is very clear for the user that there is an overlay on top and it is clear which app generated it: we build on top of this observation to further extend our proposal with the introduction of a well-defined API to create *system popups*. We define a system popup as a system-generated, clearly-defined overlay that resembles web-based popups: the shape and layout are defined by the OS and cannot be customized by an unprivileged app, and the only freedom left to the app developer is the definition of the popup's content and whether to show one (or more) clickable buttons. Moreover, a system popup will include information about the app requesting its creation, so to avoid any source of confusion. Last, the OS should also dim the background out, as it already happens for standard dialogs.

We note that we introduce the concept of system popups not to address a specific design shortcoming, but to address a potential usability limitation of our proposal. In a way, we remove the need for legitimate apps such as LastPass to have access to an API to create *arbitrarily custom* overlays, which, as we have seen, provide a powerful primitive to mislead users.

C. Limitations

We acknowledge that our proposal has a few limitations. First, we leave the responsibility to indicate which widgets should be considered as security-sensitive to the developer. An interesting future research direction is how to automatically determine where to place these checks. We note that this limitation affects the current system as well. Second, the proposed modifications would prevent certain types of security-sensitive widgets to generate accessibility events, which could be useful in some specific context. One solution to not negatively interfere with these scenarios is to allow only system-signed apps to have full access to these security-sensitive widgets: since system-signed apps already have very elevated privileges and are considered a trusted extension of the OS itself, we argue this is a reasonable compromise. Last, our proposal does not prevent a malicious app to use the BIND_ACCESSIBILITY_SERVICE permission as a side-channel to infer which app the user is interacting with and to subsequently mount existing GUI confusion and task-hijacking attacks [1], [2], [3]. While these attacks are not as stealthy as the ones we proposed, they could still be effective against the less security-conscious users.

D. Short-term Recommendations

Our proposal would be implemented as a series of system modifications. Thus, by its nature, it is going to take some time for our proposal to be deployed on users' devices. On the other hand, this work shows that the threat associated to these attacks is real. We argue that, in the meantime, Google should follow two recommendations.

First, the SYSTEM_ALERT_WINDOW permission can seriously hinder the security of the entire device, and we strongly believe it should not be automatically granted, not even for apps hosted on the Play Store. Moreover, our user study suggests that users do not clearly understand the security implications of an app with this permission. By not automatically granting this permission, the Android OS would have a chance to explain what an app with this permission is really capable of.

Second, we believe Google should scrutinize more accurately apps with this combination of permissions. As we discussed earlier, it was trivial to get an app requiring both

permissions and including malicious-looking functionality, such as the possibility of downloading and executing arbitrary code from a network end-point. Since the volume of submitted apps requiring both these permissions seems to be relatively low (only 17 apps out of the 4,455 top apps we crawled require both permissions), we believe that even *manual vetting* could be a scalable approach.

These recommendations would not address the design shortcomings we identified, but they have the advantage of being immediately deployable. In fact, the first one would simply require an update of the Play Store app, which Google obviously controls, while the second one simply requires more scrutiny during the vetting process.

X. Related Work

Recent years have seen the migration of UI attacks to mobile OSs. For example, Rydstedt et al. [15] demonstrate that mobile browsers are vulnerable to framing attacks, while Felt et al. [11], Niemietz et al. [12], Chen et al. [1], Bianchi et al. [2], and Ren et al. [3] study the use of UI attacks to lure users to enter their credentials into fake/malicious UIs. These works showed that these attacks are practical, and can affect millions of Android apps.

However, the previous attacks suffer from two limitations. First, they rely on OS-provided side channels (e.g., access to the `/proc` file-system), which are being systematically removed by Google. As a result, all known task hijacking techniques will not work on modern versions of Android. Second, after the user has inserted her credential into the malicious UI, the user will not see the expected result of a successful login, and she might become suspicious. This highlights the importances to have *both the ability to show the users the attacker's intended UI, and also to preserve the expected user experience once the fake UI is dismissed;* the clickjacking attacks presented in this work address these limitations by utilizing the accessibility service to know when to pop-up our fake UI and to log the user in using the stolen credential after the fake UI is dismissed (cf. Attack #5).

The "draw on top" feature has been abused by malware to mount clickjacking attacks [16], [17], [18], [19], [7], [20] (which was first introduced as "redressing" attacks by Niemietz et al. in [12]) to, for example, lure the user into enabling device administrator capabilities. In response, Google introduced the `FLAG_WINDOW_IS_OBSCURED` flag to counter the risk. Our work shows that this current defense mechanism is vulnerable *by design*. Moreover, we show how the existing clickjacking techniques can be significantly improved in multiple regards: we show how the complexity of the Android framework often leads to very fine-grained side channels, and how it is possible to implement context-aware clickjacking. Our user study also shows that these attacks are extremely practical and stealthy. Furthermore, by combining the SYSTEM_ALERT_WINDOW permission and the accessibility service, we made previous clickjacking attacks more practical by *removing the reliance on vanishing side-channels (to know when to show the fake UI) and by reliably dismissing the fake UI when the user clicks on something to preserve the user experience.*

Recently, malware have been found using Android's accessibility service to bypass security enhancements or to install apps [4], [21], [22]. Kraunelis et al. [23] point out how malicious apps can abuse the accessibility service on Android to detect app launching, and to race running on the top, while Jang et al. [24] showed that accessibility service, which is available on all popular systems, leads to a new kind of vulnerabilities in existing defense. All these existing attacks are visible to the users, thus will lead to user reactions that are detrimental to further attacks. In contrast, our work shows how one can use the SYSTEM_ALERT_WINDOW permission to distract the user and cover up the almost arbitrary malicious operations performed by BIND_ACCESSIBILITY_SERVICE underneath. Thus, for the first time, we add stealthiness to these already devastating attacks: our user study showed that *none* of the 20 human subjects even realized they were attacked.

Along with the increasingly popular attacks on mobile devices, there is also a large body of works proposing defense solutions. Malisa et al. [25] propose to detect spoofing attacks by leveraging screenshot extraction and visual similarity comparison, but our attack does not use spoofed UI. Bianchi et al. [2] propose the use of a security indicator to help users identify which app they are interacting with, and make sure that the inputs go to the app. In [26], Fernandes et al. find that this approach introduces side channels and they propose to notify the user when a background non-system app draws an overlay on top of the foreground app. However, this will disrupt many legitimate apps that use overlays (e.g., Facebook, LastPass); our proposal, instead, provides app developers the flexibility to determine when they want to prohibit overlays. Similarly, Ren et al. propose the Android Window Integrity policy [27], which restricts the use of overlay to only white-listed apps and also protects the navigation between apps/activities using the "back" and "recent" buttons.

Some of the defense mechanisms proposed in Section IX are inspired by previous works. For example, we adopted the idea that overlays cannot cover secure widgets, and users should have sufficient time to interact with them, which is proposed by Ringer et al. [28] and Roesner [14]. However, if the semantic of the secure widgets relies on neighboring widgets (e.g., a generic "OK" button), the approach proposed in [28] may suffer the same problem as the `FLAG_WINDOW_IS_OBSCURED` flag; in this case, we need to protect both the widget and the associated text. Moreover, the proposal to prevent accessibility service from programmatically interacting with secure widgets is similar to the input integrity principle from Roesner et al. [29]. Moreover, we are inspired by Jang et al. [24]'s recommendations that allow developers to flag how different widgets will handle inputs from accessibility service. Finally, it's possible to use the method from Nan et al. [30] to automatically identify security sensitive UI widgets and apply the proper defense.

XI. Responsible Disclosure & Reception

Disclosure. We responsibly disclosed our findings to Google's Android security team. We started by reporting a number of "easy-to-describe" bugs as AOSP issues [31]. The bug enabling Attack #3 was promptly acknowledged as a "Moderate severity" bug. However, to date, it is still unpatched. To our surprise, our report showing how an accessibility app can steal passwords, steal second-factor authentication tokens, and lock and unlock the phone while keeping the screen off was marked, after several back and forth, as "work as intended." We then noted that our findings clearly showed how security mechanism #4 is completely bypassed. The Android team then followed up stating that they would have updated the documentation. We noticed that the accessibility service documentation has been indeed updated, and the "Security note" has been silently removed. Given these quite unsuccessful attempts to convey the gravity of our findings via traditional bug reports, we opted to share a draft of our paper with Adrian Ludwig, director of Android security. We note that, although we have not been in direct communication with Adrian, we have been told that our paper has been read and seriously considered by the Android security team.

Patch Deployment. Unfortunately, to date (July 2017), no patch has been deployed to stop these attacks. That is, Android 6.0.1 and Android 7.1.2 are both still vulnerable to all attacks described in the paper, even after having installed the monthly security update distributed on July 5th. As mentioned above, the only observable change was to the removal of the "security note" in the documentation of the accessibility service. We believe that the delay in developing and issuing appropriate security patches is due to these bugs not being well-understood "traditional" security bugs, such as buffer overflows.

Google's Official Statement. The initial version of this work was published in the proceedings of the IEEE Symposium of Security and Privacy 2017 [32]. In addition, we setup a website to host additional information and demos [33]. This work received significant press coverage and, when contacted by some news outlets, a Google spokesperson released the following statement: "We've been in close touch with the researchers and, as always, we appreciate their efforts to help keep our users safer. We have updated Google Play Protect our security services on all Android devices with Google Play to detect and prevent the installation of these apps. Prior to this report, we had already built new security protections into Android O that will further strengthen our protection from these issues moving forward." This answer pushed us to investigate two different aspects: 1) techniques and approaches to automatically detect Cloak & Dagger-based attacks, and 2) how these attacks affect the current release of Android O. We discuss these aspects in the upcoming sections.

XII. Detecting Cloak & Dagger

We believe that automatically detecting an application implementing Cloak & Dagger attack would be quite challenging. Of course, it is possible to develop a signature-based approach, but the uncovered design issues enable a number of different attack scenarios and are difficult to generalize. One possibility would be to perform anomaly detection and monitor the behavior of the app for what concerns the UI activity, such as how many overlays the app is creating and how frequently. However, an app can delay these activities for an undetermined amount of time (to bypass Google's automatic vetting mechanisms) and it is not clear whether Google can enable this sort of fine-grained logging on the users' devices. One viable solution is to statically detect apps requiring both the permissions abused in this paper (SYSTEM_ALERT_WINDOW & BIND_ACCESSIBILITY_SERVICE) and to perform manual vetting on them especially when published by a developer without a long-standing good reputation. Since the number of apps requiring both permission appears to be quite small, manual vetting might be a scalable approach.

That being said, we put ourselves in the shoes of the attacker and tried to investigate whether our attacks could be bootstrapped by an app that requires only one or zero permissions. Our investigation led to the development of new attacks that allow a malicious app to bootstrap all the attacks starting only from the SYSTEM_ALERT_WINDOW permission (Android 6.0.1 and Android 7.1.2) and another attack that show how to implement all Cloak & Dagger attacks starting with an app *requiring no permissions* (Android 6.0.1).

XIII. Bootstrapping from One Permission

We developed a new strategy that a malicious app can use to bootstrap the attacks presented in this paper by requiring only one permission, the SYSTEM_ALERT_WINDOW. The key finding that enables this attack is the following: the official Google Play Store app itself is vulnerable to clickjacking, and it is not even protected by the "obscured flag." In particular, we found that the "Install" button (to install an app on the Play Store) and the "Open" button that appears when browsing on the Play Store page of an already installed app are both vulnerable. Thus, a malicious app with only the SYSTEM_ALERT_WINDOW can silently install a secondary malicious app (requiring *only* BIND_ACCESSIBILITY_SERVICE) by following these steps: 1) create an on-top opaque overlay, 2) open the Play Store app and browse to the page of the target secondary malicious app (this can be done programmatically, through Intents), 3) lure the user to click on the "Install" button (unknowingly, through clickjacking), 4) wait that the secondary app is installed, 5) open the Play Store app to the page of the target secondary malicious app, which will now show an "Open" button (again, this can be done programmatically), and 6) lure the user to click on the "Open" button. At this point, the secondary malicious app is installed and running on the victim's phone. The initial malicious app can now lure the user to enable the accessibility service for the *second* application.

This is approach powerful for two reasons: First, none of the two apps needs to require both permissions, and it is thus more difficult to detect them as part of the vetting process. Second, the two apps can now collaborate to implement the

attacks uncovered in this paper. Each app would contain only part of the malicious functionality, and not the entire payload, thus making them more difficult to be automatically detected. However, while this attack is stealthier, it is slightly less practical: first, the initial malicious app needs to hijack two additional clicks (but, as explained in Attack #1 & #2, these clicks do not necessarily need to be consecutive or near in time); second, the malware author needs to submit to the Play Store an additional malicious application.

XIV. Bootstrapping from Zero Permissions

We developed another variation of these attacks that can be bootstrapped by an application with zero permissions. This attack builds on the attack we discussed in the previous section, where we show how the attacks can be bootstrapped with an app with SYSTEM_ALERT_WINDOW. The question we answer here is: is it possible to remove the constraint of requiring the SYSTEM_ALERT_WINDOW permission? To our surprise, we found out that in Android 6.0.1, it is actually trivial to remove this requirement. In fact, the SYSTEM_ALERT_WINDOW permission is needed to create overlays of type `TYPE_SYSTEM_ALERT`. However, for these attacks, we do not strictly need to create overlays of that type – we just need to create overlays that are on top of all app's activities. It turns out that overlays of type `TYPE_TOAST` satisfy this requirement: all toasts are in fact shown on top of other activities. Traditionally, Toasts have been abused by repeatedly invoking the `Toast.makeText` API. This API accepts a "duration" argument, but the duration can either be 2 or 3.5 seconds. Previous work achieved "persistent Toasts" by repeatedly invoking this API in a loop. We found that it is actually even simpler to create persistent and arbitrary Toasts: one can simply use the same API that is used to create arbitrary `TYPE_SYSTEM_ALERT` windows. This API allows to bypass the duration constraint. We were able to port our proof-of-concept to use overlays of type `TYPE_TOAST` by simply running this command: "`sed -i "s/TYPE_SYSTEM_ALERT/TYPE_TOAST/" *`."

We note that the attack we discussed in this section works for Android 6.0.1, but it does not work on Android 7.1.2. The reason is that Android 7.1.2 features a number of protection mechanisms to prevent abuses of `TYPE_TOAST` overlays. We also note that a number of previous works has already shown how to abuse Toasts for several purposes. While writing this white paper, we also discovered a blog post that discusses how Toasts can be used to specifically mount clickjacking attacks [34]. To our surprise, this article is dated May 26th, 2011, when "[...] close to 96% of the Android ecosystem is still using Android 2.2 and below." The blog post also includes the following statement that is relevant to the attack discussed in the previous section, where we show how the Play Store app is completely unprotected: "We also hope that Google will begin to make use of their own security features in future releases of their own packages, such as settings, dialer, and market applications." It is quite worrisome that a security researcher noticed and notified Google several years ago and most of Google's important applications (e.g., Settings app, Play Store) are still vulnerable to clickjacking. We hope this work is going to raise more awareness and we urge Google to deploy an appropriate fix. Our user study suggests these attacks are practical and they might pose a real threat.

XV. Attacking Android O

We investigated how our attacks affect the current beta version of Android O. For our tests, we flashed Android O Developer Preview 3 (made available by Google at [35]) on a Nexus 5X. We first tried Attack #3 (the "Invisible Grid Attack"): fortunately, Android O seems to be protected by this attack. Upon investigation, it seems that the Android framework has been patched to not set the "obscured flag" for `ACTION_OUTSIDE` Motion events, as we recommended in our paper. However, it appears that all the other attacks are still applicable. Moreover, we noticed that, supposedly due to a regression, the final "OK" button to enable accessibility service (see Figure 1a) is not protected by the obscured flag. That is, Security Mechanism #3 does not seem to be deployed. This aspect makes enabling accessibility service through clickjacking even more practical on Android O than it is on Android 6.0.1 and Android 7.1.2. We emphasize that we only tried these attacks on the Preview developer version of Android O, and not on the final release. Our hope is that the final version of Android O will be protected by these attacks.

XVI. Conclusion

In this paper we have uncovered several design shortcomings that, in turn, make an Android app with the SYSTEM_ALERT_WINDOW and BIND_ACCESSIBILITY_SERVICE permissions able to mount devastating and stealthy attacks, which we called "cloak and dagger." Our experiments show that it is very easy to get an app on the Play Store and that the context-aware clickjacking, silent installation of a God-mode app, and keystroke inference attacks are both practical and stealthy. We also show that a malicious app could even bootstrap the attacks starting from only one or zero permissions (at the price of making the attack slightly less practical, due to the need of hijacking two more clicks). We also investigated how these attacks affect the current preview of Android O. We hope that our work will raise awareness of the real danger associated with these two permissions, and that Google will reconsider its decisions and adopt our proposed defense mechanism.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable feedback. We would also like to thank Billy Lau, Yeongjin Jang, and, once again, Betty Sebright and her growing team. We would also like to thank Google anti-malware and a11y teams for their availability to discuss our findings in more detail. This research was supported by the NSF awards CNS-1017265, CNS-0831300, CNS-1149051 and DGE-1500084, by the ONR under grants N000140911042 and N000141512162, and by the DARPA Transparent

Computing program under contract DARPA-15-15-TC-FP-006. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, ONR, or DARPA.

REFERENCES

[1] Q. A. Chen, Z. Qian, and Z. M. Mao, "Peeking Into Your App Without Actually Seeing It: UI State Inference and Novel Android Attacks," in *Proc. of the USENIX Security Symposium*, 2014.

[2] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna, "What the App is That? Deception and Countermeasures in the Android User Interface," in *Proc. of the IEEE Symposium on Security and Privacy*, 2015.

[3] C. Ren, Y. Zhang, H. Xue, T. Wei, and P. Liu, "Towards Discovering and Understanding Task Hijacking in Android," in *Proc. of USENIX Security Symposium*, 2015.

[4] Lookout, "Trojanized adware family abuses accessibility service to install whatever apps it wants," https://blog.lookout.com/blog/2015/11/19/shedun-trojanized-adware/, 2015.

[5] S. Week, "Overwhelming Majority of Android Devices Don't Have Latest Security Patches," http://www.securityweek.com/overwhelming-majority-android-devices-dont-have-latest-security-patches, 2016.

[6] "Documentation for `SYSTEM_ALERT_WINDOW` (`DRAW_ON_TOP`, informally) permission." [Online]. Available: https://developer.android.com/reference/android/Manifest.permission.html#SYSTEM_ALERT_WINDOW

[7] Y. Amit, "Accessibility Clickjacking The Next Evolution in Android Malware that Impacts More Than 500 Million Devices," https://www.skycure.com/blog/accessibility-clickjacking/, 2016.

[8] "Documentation of AccessibilityEvent." [Online]. Available: https://developer.android.com/reference/android/view/accessibility/AccessibilityEvent.html

[9] "Documentation of Accessibility's FLAG_RETRIEVE_INTERACTIVE_WINDOWS flag." [Online]. Available: https://developer.android.com/reference/android/accessibilityservice/AccessibilityServiceInfo.html#FLAG_RETRIEVE_INTERACTIVE_WINDOWS

[10] Google, "Google Authenticator App," https://play.google.com/store/apps/details?id=com.google.android.apps.authenticator2&hl=en.

[11] A. P. Felt and D. Wagner, "Phishing on Mobile Devices," in *Proc. of IEEE Workshop on Web 2.0 Security & Privacy (W2SP)*, 2011.

[12] M. Niemietz and J. Schwenk, "UI Redressing Attacks on Android devices," *Black Hat Abu Dhabi*, 2012.

[13] A. S. Team, "Android Security Bulletins," https://source.android.com/security/bulletin/, 2016.

[14] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan, "User-driven Access Control: Rethinking Permission Granting in Modern Operating Systems," in *Proc. of the IEEE Symposium on Security and Privacy*, 2012.

[15] G. Rydstedt, B. Gourdin, E. Bursztein, and D. Boneh, "Framing Attacks on Smart Phones and Dumb Routers: Tap-jacking and Geo-localization Attacks," in *Proc. of the USENIX Conference on Offensive Technologies*, 2010.

[16] W. Zhou, L. Song, J. Monrad, J. Zeng, and J. Su, "The Latest Android Overlay Malware Spreading via SMS Phishing in Europe," https://www.fireeye.com/blog/threat-research/2016/06/latest-android-overlay-malware-spreading-in-europe.html, 2016.

[17] T. Seals, "Autorooting, Overlay Malware Are Rising Android Threats," http://www.infosecurity-magazine.com/news/autorooting-overlay-malware-are/, 2016.

[18] T. Spring, "Scourge of android overlay malware on rise," https://threatpost.com/scourge-of-android-overlay-malware-on-rise/117720/, 2016.

[19] M. Zhang, "Android ransomware variant uses clickjacking to become device administrator," http://www.symantec.com/connect/blogs/android-ransomware-variant-uses-clickjacking-become-device-administrator, 2016.

[20] Y. Amit, "95.4 percent of all android devices are susceptible to accessibility clickjacking exploits," https://www.skycure.com/blog/95-4-android-devices-susceptible-accessibility-clickjacking-exploits/, 2016.

[21] S. Lui, "Accessibility Service Helps Malware Bypass Android's Beefed Up Security," http://www.lifehacker.com.au/2016/05/accessibility-service-helps-malware-bypass-androids-beefed-up-security/, 2016.

[22] D. Venkatesan, "Malware may abuse androids accessibility service to bypass security enhancements," http://www.symantec.com/connect/blogs/malware-may-abuse-android-s-accessibility-service-bypass-security- enhancements, 2016.

[23] J. Kraunelis, Y. Chen, Z. Ling, X. Fu, and W. Zhao, "On Malware Leveraging the Android Accessibility Framework," in *International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*, 2013.

[24] Y. Jang, C. Song, S. P. Chung, T. Wang, and W. Lee, "A11y Attacks: Exploiting Accessibility in Operating Systems," in *Proc. of the Conference on Computer and Communications Security (CCS)*, 2014.

[25] L. Malisa, K. Kostiainen, and S. Capkun, "Detecting Mobile Application Spoofing Attacks by Leveraging User Visual Similarity Perception," in *Cryptology ePrint Archive, Report 2015/709*, 2015.

[26] E. Fernandes, Q. A. Chen, J. Paupore, G. Essl, J. A. Halderman, Z. M. Mao, and A. Prakash, "Android UI Deception Revisited: Attacks and Defenses," in *Proc. of Financial Cryptography and Data Security (FC)*, 2016.

[27] C. Ren, P. Liu, and S. Zhu, "WindowGuard: Systematic Protection of GUI Security in Android," in *Proc. of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2017.

[28] T. Ringer, D. Grossman, and F. Roesner, "AUDACIOUS: User-Driven Access Control with Unmodified Operating Systems," in *Proc. of the Conference on Computer and Communications Security (CCS)*, 2016.

[29] F. Roesner and T. Kohno, "Securing Embedded User Interfaces: Android and Beyond," in *Proc. of the USENIX Security Symposium*, 2013.

[30] Y. Nan, M. Yang, Z. Yang, S. Zhou, G. Gu, and X. Wang, "UIpicker: User-Input Privacy Identification in Mobile Applications," in *Proc. of the USENIX Security Symposium*, 2015.

[31] Google, "AOSP Issue Tracker," https://issuetracker.google.com/.

[32] Y. Fratantonio, C. Qian, P. Chung, and W. Lee, "Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop," in *Proc. of the IEEE Symposium on Security and Privacy*, 2017.

[33] ——, "Cloak & Dagger Website," http://cloak-and-dagger.org.

[34] K. Johnson, "Revisiting Android Tapjacking," https://nvisium.com/blog/2011/05/26/revisiting-android-tapjacking/, 2011.

[35] Google, "Factory Images for Android O preview," https://developer.android.com/preview/download.htmlflash, 2011.

APPENDIX

#	Attack Name	Attack Primitives	Design Shortcoming	Requested Permission	Malicious Functionality
1	Context-aware Clickjacking	① ② ④	① ③ ④	DRAW	Clickjacking attacks aware of users' actions
2	Context Hiding	①	① ④	DRAW	Hide real context a user is interacting with
3	Keystroke Inference	④	① ③	DRAW	Record all users' keystrokes
4	Keyboard App Hijacking	④	②	ACCESS	Record all users' keystrokes
5	Password Stealing	② ③ ④	① ④	DRAW & ACCESS	Steal users' credentials
6	Security PIN Stealing	④	②	ACCESS	Steal users' PIN
7	Phone Screen Unlocking	③	②	ACCESS	Unlock the device's screen
8	Silent App Installation	① ③	① ② ④	DRAW & ACCESS	Install apps silently
9	Enabling All Permissions	① ③	① ② ④	DRAW & ACCESS	Enable apps' permissions silently
10	2FA Token Stealing	②	② ④	ACCESS	Steal all users' tokens
11	Ad Hijacking	② ④	① ④	DRAW & ACCESS	Hijack other apps' ads clicks
12	Exploring the Web	③	② ④	ACCESS	Open the browser, browse arbitrary pages, and perform arbitrary operations

TABLE I: Systematization of the attacks. For each attack, we report the attack primitives they rely on, the design shortcoming they exploit, the permissions they request, and a short summary of the malicious functionality they implement. Note that, for conciseness, we indicate SYSTEM_ALERT_WINDOW with DRAW and BIND_ACCESSIBILITY_SERVICE with ACCESS.