2016 IEEE 40th Annual Computer Software and Applications Conference

Maintaining User Interface Integrity on Android

Abeer AlJarrah
College of Computing & Informatics
University of North Carolina at Charlotte
Charlotte, North Carolina 28223
Email: aaljarra@uncc.edu

Mohamed Shehab
College of Computing & Informatics
University of North Carolina at Charlotte
Charlotte, North Carolina 28223
Email: mshehab@uncc.edu

***Abstract*—The demand of having a multi-window and multi-tasking option in Android devices has been an emerging topic among Android users, especially with the trends toward larger hand-held screen sizes. One option to meet this demand, is to use floating windows. This feature enables users to perform more than one task at the same time while sharing the same screen. Device screens can be divided into multiple windows that can have different visual features in terms of size, location and transparency. While this feature addresses user complaints about Android on large screen devices, attention must be given to the security implications of such an option.**

In this work, we demonstrate how the current implementation of floating windows on Android can be abused to compromise user interface integrity through several attacks such as tapjacking, event eavesdropping and eventhijacking.

Although previous versions of Android have evolved to handle the issue of eventhijacking enabled by `Toasts`, recent versions fail to address security concerns related to floating windows. We propose and describe two approaches, an application level and a system level, to enable secure apps against possible malicious floating windows. The application level approach aims to detect existence and location of floating windows on top of an app. System level approach not only detects their existence, but also extends the system to include an event handler that notifies apps when floating windows are rendered over the apps' secure regions. We implemented our proposed approaches and performed experiments to evaluate their efficiency.

I. Introduction

Floating Apps, floating windows or floating views are all terms that interchangeably refer to apps with the ability of having more than one active application window running on the mobile display screen. They are advertised as "cool" apps that have one or more windows floating over other running applications allowing real-time multitasking behavior. Users can enjoy the desktop-like experience on their mobiles, since they can move, re-size, maximize and minimize windows of running apps. Users can also pin installed applications into floating views, enjoying multitasking and fully utilizing the whole screen. These windows are basically views which do not occupy the whole screen yet provide complete functionality such as playing music or chatting with friends. Users can have one or more active windows on the screen while still being able to interact with home screen or a background app. Floating windows are shipped to the market in many forms. They can be found in custom ROMs (Read-Only Memories) such as Paranoid [1], or packaged as a feature of a framework that needs root access such as Xposed Framework [2], or simply through libraries that neither require installing a custom ROM nor root access, such as StandOut library [3]. This is an attractive feature for users who have large screen smartphones and tablets. Fig. 1 shows a screenshot of one of the floating apps from Google Play.

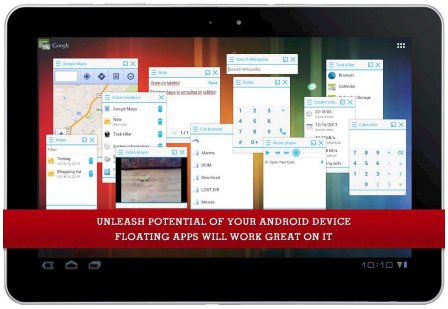


Fig. 1: Floating Views apps [4]

Although having floating views or windows provide multitasking functionality, it comes with potential security implications. For example, a floating window may attempt to steal input intended for background app or forge user's intent to interact with it by tricking the user. Furthermore, it is possible to have floating windows that are stealthily overlaying critical input fields or action items on the background app. These threats are collectively referred to User Interface Redressing (UI Redressing), which basically involves tricking the user into interacting with the UI element that is different than the intended one. UI Redressing has been investigated in the web context, where `<iframe>` or `<div>` can be used as an overlay. Many attacks and solutions were investigated in that context [5] [6] [7] [8] [9]. However, this paper is the first to discuss the UI overlay security on the context of floating views in a native mobile platform.

The Android Security model has evolved over time to address similar security threats. One example is View class XML attribute `android:filterTouchesWhenObscured` and method`setFilterTouchesWhenObscured(boolean)`., which were added to Android versions 2.3 and above. The purpose was to specify whether to filter touches when the view's window is obscured by another visible window. This was in light of the security bug reported exploiting the TapJacking vulnerability enabled by `Toasts` [10] [11]

0730-3157/16 $31.00 © 2016 IEEE
DOI 10.1109/COMPSAC.2016.150

[12]. `Toasts`, unlike other view types, pass touch events to underlying views which can be abused maliciously. Adding these attributes to the View class helped prevent a considerable attack vector. Nevertheless, threats posed by overlays of window types other than `Toasts` are still a possible vulnerability. In fact, compromising smartphone display in general may have serious implications that are not applicable to web-based applications or desktops. For example through compromising UI on mobile devices, it is possible to infer where users tap and what they type on a smartphone display or tablet display based on sensors data such as accelerometer, gyroscope readings or orientation sensors [13] [14]. Today's system developers who wish to protect their apps against threats introduced by a floating window have no systematic support from the platform where they can define a custom behavior to protects user's data. In this paper, we propose two approaches to handle floating windows. We extend the implementation of Android 4.4 (KitKat) to add the functionality of defining a custom behavior in case the app was drawn under a floating window. We also, propose an application level approach to help detect malicious overlays without the need of changing Android core. This paper is organized as follows: Section II discusses previous and related work. Then, Section III presents the threat model in light of the current platform implementation. Section IV provides background information by illustrating how Android manages UI and how overlays are designed and used. Section V presents the proposed approaches to help countermeasure possible attacks. Lastly, we provide some concluding remarks in Section VI.

II. Literature Review

Attacks targeting UI were extensively studied in web context; terms like clickjacking and UI Redressing were initially addressed as pure web-based terminology. These attacks' viability were studied later in native OS context in terms of how they can be mounted, consequences and possible countermeasures. We will briefly shed light on related work of UI Attacks in different contexts.

General Web-based UI Attacks In 2008, Clickjacking was first introduced [15] as an attack that compromises browsers' user interface integrity by utilizing `<iframe>` to hold a page of interest to the attacker then place it over a page that the user actually sees, luring the user to click on a link that will actually be received by the framed page. Extensive research has been conducted to solve this problem. Frame busting, which is simply JavaScript code that pulls the current layer into the top, is one the first techniques introduced to solve this issue. However, studies [5] showed browsers at that time can be easily circumvented even with frame busting code. HTTP Headers like `X-FRAME-OPTIONS` option were introduced to give developers options to either prevent browsers from rendering iframes or check if it is loaded from the same origin. Other solution was to simply use no script , which means that browsers will not render any Javascript. Another solutions suggested on the client side rather than server side were presented in the form of browser extensions [8] [16]. These basically used clickjacking attack attributes like visibility. Counter actions by these plugins ranged from blocking Javascript, randomizing UI or simply warning the user. To overcome limitations of server and client side solutions and address more complex attacks, proxy-level framework [7] was designed to check for symptoms of clickjacking attacks by analyzing requests and response pages and removing malicious attack payloads to safeguard end users. In 2012, Huang et al [6] revisited existing clickjacking defense mechanisms and showed that they are not sufficient. They designed new clickjacking techniques based on existing ones and devised *InContext* defense mechanism. In their study, they explained that the root cause of clickjacking is that malicious applications appear as out of context UI elements to the user, which makes the user act out of context as well. To keep user actions *in context*, they advised that websites should mark their critical UI elements and then let browsers enforce context integrity of user actions on the sensitive UI elements. This approach works by enforcing visual and temporal integrity. Visual integrity can be achieved through guaranteeing target display integrity and pointer integrity. Temporal integrity, which focuses on giving the user enough time to conceive her action's consequences, can be achieved through methods of UI delay. UI delay after pointer entry padding area around critical elements and pointer re-entry on newly visible sensitive element.

In their work [9] Akhawe at al., not only shed light on how human perception limitations relate to click-jacking attacks, they have also developed attack vectors that use these limitations existing defense mechanisms fail to circumvent. They have also stressed that badly designed usability is the main reason why most of the existing defense mechanisms are not widely adopted. Not to mention that they also fail to address equivalent threats on touch-based devices.

Mobile Web-based UI Attacks Rydstedt et al. studied framing attacks' viability on the various browsers of smartphones [17], and their findings showed that not only most mobile websites do not implement proper busting but also smartphone specific features like iPhone zooming, sharing screen real-estate with the browser and others can be easily abused to form what they called "tap-jacking" attack.

In their effort to explain how smartphone webviews specifically are vulnerable, Lou at al. [12] acertain that WebViews APIs can be abused by malicious developers to mount what they referred to as a "touchjacking attacks". They have explained how the UI APIs can help attackers control WebView position, events , display and layout.

Android specific UI Attacks The fact that `Toasts` allow touch gestures to be passed to beneath surfaces has been discussed as one of the first vulnerabilities that can be used to mount touch-jacking attacks [18]. The attack basically depends on displaying to what appear as a benign notification (`Toast`) in the foreground while there is a running malicious application in the background. Android versions after 2.3 provided a capability protection mechanism using `android:setFilterTouchWhenObscured()` or alter-

natively `android:filterTouchWhenObscured`; however, these properties are available for versions after 2.3, not to mention that they are not enabled by default. Later work by Niemietz and Schwenk [11] demonstrated what they refer to as "browser-less" tapjacking attacks using transparent action UI items such as `Buttons` added *above* a victim app using `WindowManager`. Several attack permutations based on this concept can compromise critical applications such as contacts, native browser, touch gesture logging, premium calls and installing applications in background without user consent. The work suggested a theoretical solution which basically relies on instantiating a layer (*Tapjacking Security Layer* or TSL) beneath the application. This layer instantiates once the app is launched so that it blocks all the touch gestures from reaching any further. TSL should remain open until the application closes. Although this work [19] does not address clickjacking explicitly, it highlights a flaw in Android design that can leak UI state information through their discovered side channel. An attacker first builds a UI state machine based on UI state signatures constructed offline, and then infers UI states in real time from an unprivileged background app. To circumvent this issue, several defense mechanisms were suggested, such as having more strict access control on memory files and change the way WindowManager buffer windows.

In the context of using UI to detect malicious behavior, Huang et al, [20] proposed to define a malicious behavior as a mismatch between UI and app behavior. They extracted UI elements texts and compared connotations behind captured text to the functions or APIs being called.

In the context of securing embedded UI on Android, Roesner & Kohno [21] have redesigned Android to support secure embedded interfaces on both OS and web based context (LayerCake). LayerCake contains two main additions, `EmbeddedActivityView` and `SecureWebview`. `EmbeddedActivitView` extends Android activity class to securely host embedded UI components such that each embedded view resides on a separate window that overlays the main window (parent activity) on a nested manner. This separation of windows prevents attacks like parent eavesdropping, DoS or clickjacking on child windows, and also prevent child view from manipulating parent UI. LayerCake supportes clickjacking prevention by allowing embedded activity to block user input if the activity is covered by a `Toast`, doesn't have minimum required size or is not fully visible.

LayerCake also implementes `SecureWeView`, which is an activity that contains a web view but each one resides on a separate window. When `SecureWebView` is embedded in another activity, the internal web view takes the dimensions of the hosting activity. Because of the separation of parent and child into two activities, i.e. two processes, it is no longer possible to eavesdrop input, extract input, issue events or manipulate size, transparency or location.

To our best knowledge, security implications using floating windows have not been discussed by any previous work. Our work is a first step towards understanding the risk associated with such feature.

```
<LinearLayout
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="#80000000"
    android:orientation="vertical"
    android:padding="8dp" >

    <EditText
        android:id="@+id/userName"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textColor="#ffffff"
        android:textStyle="bold" />

    <EditText
        android:id="@+id/passWord"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:inputType="textPassword"
        android:textColor="#ffffff" />
</LinearLayout>
```

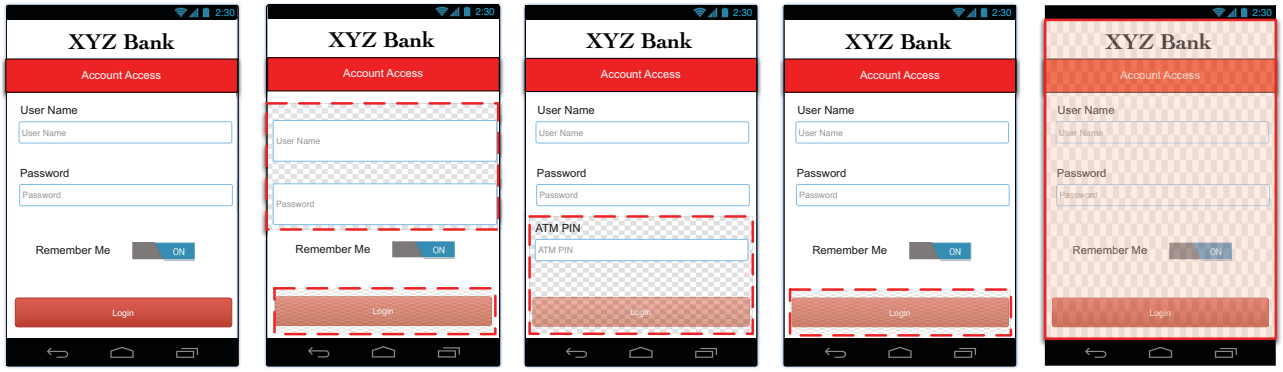
Fig. 2: Layout for transparent window

III. Security and Threat Model

Floating view or window can be displayed by instantiating an instance of `FrameLayout` which is a descendant of the class `ViewGroup`. The instance can inflate a static XML layout through a `View` object. Opacity can be controlled by XML view attributes. See Fig. 2, which shows a layout for a transparent window that have two input fields which can be used to cover login controls. `WindowManager` can be used to add/ remove instances of windows through the methods `addView()` and `removeView()`.

The main risk associated with having overlays is when they take control over the display screen on the phone surreptitiously. Malicious apps may target specific critical apps on the victim's device displaying invisible overlays that are designed specifically for the UIs of targeted apps and seamlessly display overlays that mask certain areas of the targeted UI. This can be implemented using a `Service`, which will be responsible of monitoring running apps and displaying the overlay windows accordingly. The overlay windows can appear once and then disappear after performing a malicious act such as stealing critical user input. Malicious developers might decide to destroy the invisible window in order to minimize effect on normal app behavior, thus minimizing user attention to any unusual behavior. There are many ways to compromise apps' security by hacking the display screen. Fig. 3 shows different attacks on an app UI. For the sake of randomization, we use XYZ bank app in Fig. 3a as an example of a victim app.

User Input Theft: This attack uses a transparent overlay to steal user credentials, while deceiving the user into believing that a legitimate UI input field received it. In fact, this opens the door for different permutations of well-known user input theft attacks such as user credentials theft as shown in Fig. 3b. Customized overlays can be aligned perfectly to the user input fields, preserving all the visual properties of the screen so that they seamlessly overlay the critical input fields. The user will enter her credentials into the input fields thinking that she is interacting with the app UI elements while the top transparent overlay receives the input. The attacker can know when the user hit the login button by overlaying the



(a) Victim App (b) User Credentials Theft (c) Field Display Injection (d) Touchjacking (e) Denial Of Service Attack

Fig. 3: Possible Overlay Attacks

button. This will help the attacker displaying some message to the user to enter the credentials again. After stealing user input, the attacker may hide the overlay windows. This will not raise any concerns at the user's end since she will think she mistyped the input fields and login again as usual.

Fields Injection: Floating overlays can also steal other kinds of sensitive data from the user. This can be done by displaying a floating window that contains additional input fields requesting the user to fill in order to login. This fake input field on the top of a victim's login screen will trick the user into divulging sensitive data such as ATM PIN or a SSN. Fig. 3c shows an example of a compromised app used to trick the user to input their ATM PIN. This attack is analogous to code injection attacks (XSS) in the sense of manipulating UI by adding illegitimate input fields. In XSS case, the HTML DOM variable is manipulated by script injection [22].

Clickjacking Attack: Clickjacking is a malicious approach that aims to deceive users, clicking something different from what they observe. This can be achieved through floating overlays by having transparent views floating on the top of any clickable item on the victim screen, see Fig. 3d. A transparent overlay can be positioned on the top of clickable items in the app UI. The windows are approximately positioned to cover the action item on the victim app. When the user clicks on login button thinking she is accessing her account, she actually clicked on the malicious overlay which can result in starting any undesired action such as sending data to remote server.

Denial Of Service Attack (DoS): It is possible for a running service to create an overlay spanning the entire screen, preventing any user input from going through (see Fig. 3e). When the developer (or attacker) chooses to start the service at boot time, this effectively constitutes a DoS, since the layer will obstruct the interaction with the app.

Although floating overlays are considered important for enabling user to have more than one app running in the foreground simultaneously, it still raises many security concerns, as described earlier. Furthermore, when they are used and shipped to the market, there is no precise explanation to the user about their usage and purpose. Using floating windows requires a single android permission (`SYSTEM_ALERT_WINDOW`), which is usually presented to the user under the 'Other' permissions category or at best using vague description such as "permission to draw over other apps ", which makes it easy to be neglected by the user.

Given their rising popularity in many useful and popular apps, it is not practical to disable floating windows altogether. Still, the platform should at least provide a mechanism by which the developer can define custom actions to protect the user in case there was a floating overlay covering the app window.

IV. Background

The issue addressed in this work is directly related to Android User Interface. This section is dedicated to illustrating the mechanics behind Android UI management process. The section starts by defining the terms that will be used to explain UI display management. Then we will discuss the UI display management in Android and will explain what are floating windows and how they can be used by developers.

A. Glossary

An *Activity* is one of the main components of an Android app. An activity represents an app screen and hosts the logic and interface for this screen. Most activities occupy the whole screen display.An application may contain one or more activities. Activities get stacked according to user actions, where each pile of activities that belongs to one app is called *task* [23]. As a user navigates through the activity, it passes through four states [24]. *Active or Running* is when the activity starts and appears in the foreground and receives input focus. When the activity is partially covered by any visual component such as an alert dialogue it gets *Paused*, which is the second state. This is where the activity is not the main user input focus. When the activity is fully covered by another activity it *Stops* which means that user input focus is transferred to another activity. Lastly, an activity gets *Killed* or *Dropped* when the instance doesn't exist in the memory anymore. The *Activity Manager*, is one of the main services in Android

application framework. Its main task is to manage all activities running on the system. This includes managing communication between the activities and other services, broadcasting intents and managing activities states. The *Surface* is an object holding pixels that are being composited into the screen or simply a drawing buffer. Each window you see on the screen (a dialog, full-screen activity, the status bar) has its own surface that it draws on, and *Surface Flinger* renders these to the final display in their correct Z-order. The *Window* in Android is analogous to a window on the desktop. Each *Window* has a single *Surface* in which the contents of the window is rendered. An application interacts with the *Window Manager* to create windows; the *Window Manager* creates a *Surface* for each window and gives it to the application for drawing. The application can draw UI design in the *Surface*; to the *Window Manager* it is just an opaque rectangle. This rectangle has a single view hierarchy. The *Window Manager*, is a system service, which is responsible for many functions including: managing the Z-ordered list of windows, which windows are visible and how they are laid out on screen, managing input method window, and managing wallpaper window. Among other things, the *Window Manager* automatically performs window transitions and animations when opening or closing an app or rotating the screen. So, at any point in time, the *Window Manager* is responsible for managing multiple windows on the screen. For example, the method `relayoutWindow()` in class `WindowManager` is called whenever there is any change to the screen display (i.e. add window, re-size window, move window). The *Window Manager* includes the APIs that enable a floating window that does not occupy the whole screen to be added to an activity [25]. In Android there are 23 different types of windows, see Fig. 4, which shows the most common ones. Each type is used for a specific task and has its own properties. One of the properties that distinguishes window types is the priority in the Z-index of the display screen which decides which window gets focused and thus receives user input.

Fig. 4: Android Windows

A *View* is an interactive UI element inside of a window. The `Button` and `TextView` are examples of Views. Views are organized in a hierarchy to represent an app's layout. A window has a single view hierarchy attached to it, which provides all of the behavior of the window. Whenever the window needs to be redrawn (such as when a view has been invalidated), this is done into the window's Surface. The Surface is locked, which returns a Canvas that can be used to draw into it. A draw traversal is done down the hierarchy, handing the Canvas down for each view to draw its part of the UI. Once completed, the Surface is unlocked and posted so that the drawn buffer is swapped to the foreground to then be composed on the screen by Surface Flinger. View is responsible for managing focus and key events. The *ViewRoot* is the top of a view hierarchy, implementing the needed protocol between View and the Window Manager. The `ViewRoot.performTraversals()` method is responsible for drawing the Activity's view hierarchy into ViewRoot's offscreen surface. This function always executes inside of the UI thread's context. It is called every time a widget or layout manager calls its invalidate or requestLayout methods. ViewRoot is also responsible for managing, collecting and dispatching user input.

B. Android UI Display Management

Activity creation and layout management spans different layers and requires communication between different processes. Fig. 5 demonstrates the sequence of handling different processes invocations and their communication in order to display an activity. Assume Activity Manager and Window Manager run in Process A, App runs on Process B and Surface Flinger runs in Process C. When a new Activity is launched (user started an app activity), Activity Manager requests corrsponding activity thread in Process B to call series of functions that lead to creating the `ViewRoot` of the activity and then add a new window for that activity. The newly created window is then registered in the Window Manager. The ViewRoot also calls the Activity's `relayoutWindow` method which then calls WindowManager's `relayoutWindow` method which fetches the surface created by Window Manager Service into app process (Process B). The Window Manager also creates `WindowState` which holds display information about this particular window. The ViewRoot then, calls the `performTraversals()` method which will call the `Surface.lockCanvas()` method in `SurfaceFlinger` (Process C). This method is responsible for mapping the raw surface memory into the app's process (Process B). Communication between different processes is done using handlers and Binders IPC, tokens and flags are used to distinguish corresponding instances.

Different Activities may have one or more windows. Activities are stacked in the Activity Manager service. Furthermore, each Activity process corresponds to `ActivityRecord` object. Similarly, Windows are stacked in Window Manager service where every window is represented as a `WindowState`. There are different Window types, depending on the con-

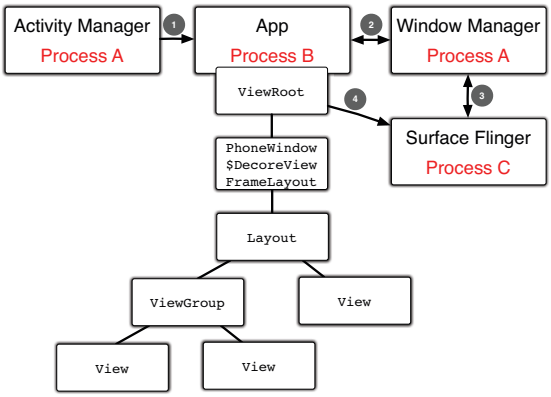


Fig. 5: Android UI Management Walk-through

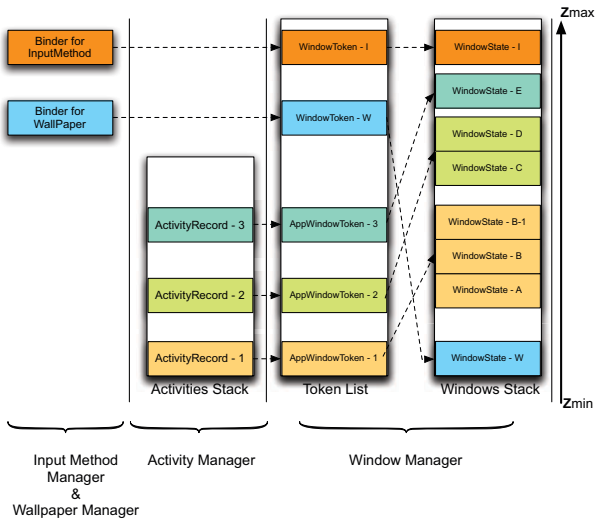


Fig. 6: Android Activities/ Windows organization

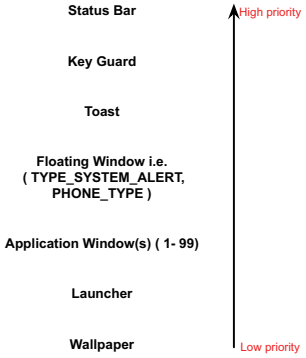


Fig. 7: Window Types Priority

tent and the purpose of the window, i.e. application activity window is different than the input method window or the wallpaper window. An Activity module may contain one or more windows, for instance if the activity shows a start-up window, or displays a pop-up dialogue, it will create a sub-window that belongs to the outer window. These windows all share the same `AppWindowToken`. As shown in Fig. 6, in WindowManager service, each activity record object corresponds to an `AppWindowToken` object. `AppWindowToken` extends `WindowToken`, which is a special type of Binder token that the WindowManager uses to uniquely identify a window in the system. Window tokens are primary security mechanism implemented to prevent malicious applications from drawing on top of the windows of other applications and share the same Window Token. The WindowManager protects against this by requiring applications to pass their application's WindowToken as part of each request to add or remove a window. If the new window token and the application token don't match, the WindowManager rejects the request and throws a BadTokenException. Without window tokens, this necessary identification step wouldn't be possible and the WindowManager wouldn't be able to provide such app isolation. The `AppWindowToken` also enables the ActivityManager to make direct requests to the WindowManager. For example, Activity Manager can request hiding all windows that belong to a specific WindowToken. WindowManager, in turn, will be able to correctly identify the set of windows which should be closed [26].

Fig. 6 shows that the Activity corresponding to **Activity Record -3** is the current system "active" activity window since WindowState-E is the top most one. ActivityRecord-1 has three window states, WindowState-A, WindowState-B and sub-window WindowState-B-1. Binders for Input method and Wallpaper corresponds to Tokens and have their own windows as well. Windows are stacked according to their Z-access position in the display screen from lowest to highest.

At any point, a screen may contain multiple app windows. At run-time, the Window Manager assigns app windows priority values ranging from `FIRST_APPLICATION_WINDOW` to `LAST_APPLICATION_WINDOW`, with values of 1 to 99 respectively. This helps the Window Manager determine in which layer these windows should be displayed. Hence, determines which window will have priority receiving user input. Z-order or layer position, determines which window receives user input from a set of windows of the same window type. For different window types, there are rules that hold that determines in which layer the window should be drawn, and the priority order in which they will receive user input.

Fig. 7 shows a subset of the window types in android and how they relate to each other in regard of the priority to receive user input. Note that Floating Window types (TYPE_SYSTEM_ALERT, PHONE_TYPE) have higher priority over application windows. This means that regardless of the time of window creation, floating windows will have higher Z-index, i.e. a higher layer than any window of type Application Window, thus, they have higher priority to receive user input first.

C. Floating Windows in Android

Creating a floating window and maintaining its presence can be done by implementing a service and then inflating a view on the screen. To add a view to the screen, developer can call `WindowManager.addView()` and specify `PHONE_TYPE` or `TYPE_SYSTEM_ALERT` as a `WindowManager.LayoutParameter` which determines the window type. These types are designed to flow on top of other application windows for the purpose of supporting system-level interaction with the user, but developers use them in their apps to get "floating containers" for their views. Layouts can be designed to include widgets and a complete logic can be implemented for these widgets event handlers. This means that these floating windows can provide functionality similar to using regular activity. Floating windows also provide a visible interactive app that doesn't occupy the whole screen.

V. Proposed Solution

The presented attack scenarios have assumed a transparent window, which means that the user is tricked because she can not see the actual surface that is receiving her input. Both of the following mitigation approaches merge in the direction of detecting if there exist an invisible floating window; hence, notify the user to take proper action.

A. Application level: Window Punching

Each Android component can be either public or private. Public components can interact with other components, but private components can only interact with those that are part of the same app (or one that runs with the same UID) [27]. In this multi-component system, security is maintained through a combination of uid-based permissions and binder capabilities. For permissions, every incoming binder transaction has associated with it the uid of the initator that is allowed to access a specific feature [28]. Otherwise, i.e. if initiator have a UID different than the component, security exception will be triggered by the OS. Based on that, and given that potential malicious floating views app and victim app are two separate applications with two different UIDs. This approach is based on injecting touch events (punches) on the current activity which, in turn should receive *all* the touch events, otherwise, it would imply a floating window on the top of the running app. We have developed a `SecureActivity` that extends the class `Activity`. This activity have a layout file that consists of one View instance covering the entire activity to act like a *cover*. Also, the extended activity has method `checkSecure()` which will switch layouts between the regular layout of the app, and the cover layout that spans the whole activity, see Fig. 8. Then a sequence of touch events is fired programmatically trying to collide with a floating window in a process we call *Window Punching*(Fig.9). Ideally, the cover view should receive all the punches (touches). Unless, there exist a floating window on the top of it, then, the OS will fire `SecurityException: Injecting to another application requires INJECT_EVENTS_PERMISSION`.

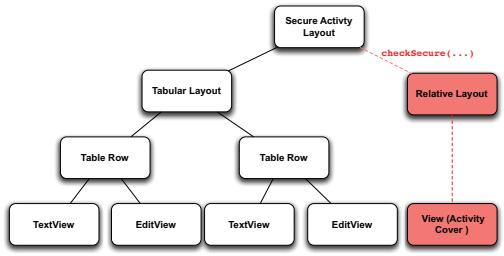
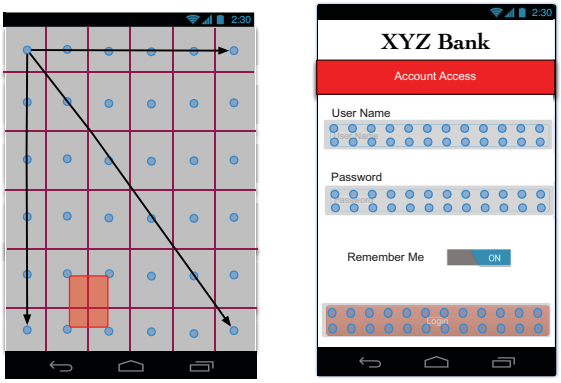


Fig. 8: Switch to cover View



(a) Whole Screen Punching (b) Selective Punching

Fig. 9: Window Punching

Punching techniques may vary in terms of location and order of punching. Whole screen punching as shown in Fig.9a punches the whole display screen. In that case the function `checkSecure(int n)` creates a mesh of size $(n \times n)$,where n is a parameter passed by the user. The mesh virtually helps decide the touch position, it contains cells where clicks are fired into. The parameter helps the user set the accuracy level of the *punching*. Hence, increasing n means increasing the number of *punches*, thus, the possibility of catching small invisible floating windows. As the parameter *n* indicates how many windows the screen will be divided into. It also implies the region size that will not receive punches, see the red window in Fig.9a. If floating window falls inside the red window, it will not be discovered.

The order of punching may have variant permutations, row-by-row , column-by-column, diagonally or simply randomly. The developer may prefer to perform punching on critical regions rather than the whole screen. Input fields and action items such as Buttons are most likely to be covered by an invisible window. Thus, developer may pass the location and dimensions of critical UI regions information to `checkSecure(Dimensions[], int n)` to perform *Selective Punching* as shown in Fig.9b. The first parameter is an array of critical regions dimensions and the second parameter is the mesh size or the accuracy of punching.

This approach can also be used to approximate the position

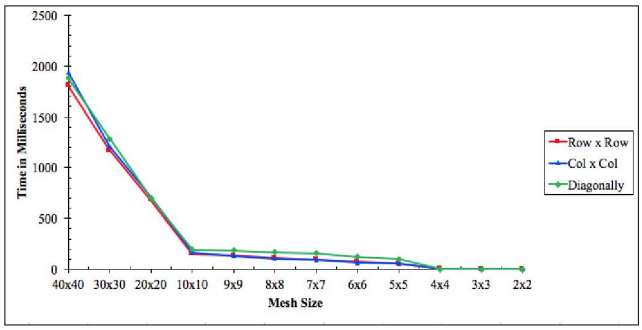


Fig. 10: Window punching performance

of the overlay through detecting the region that didn't receive the touch event(s).

To inject touch events, we have used a public API that requires no special permissions or rooting, called `Instrumentation` [29], which is used Android Automated Testing platforms such as Robotium [30]. This API can send pointer touch events using the method `sendPointerSync()`. The cover view, on the other hand, handles the touch events by logging the coordinates of the touches relative to the application window. This approach is similar to the Automated Clickjacking detection on web-pages proposed by Balduzzi et al. [8]. Their approach consisted of two units, detection and testing. The first unit was responsible for logging and detecting UI elements overlaying each other, while the second unit performed clicking and scrolling programmatically to check all clickable UI elements on HTML pages.

Performance Testing As discussed earlier, punching can be performed on the whole screen or on selective parts where critical action items reside. In this experiment, performance will be measured by extensively punching the whole screen in different orders, Row-by-Row, Column-by-Column and Diagonally. In order to see how this approach is adding overhead, we performed series of testing with varying mesh sizes. Fig. 10 shows execution time needed using different mesh sizes (2x2) to (40x40). The method `checkSecure()` contains all the heavy work of view injection and clicking all over the cover view. Touch event injection is done using multiple child threads managed by a `Threadpool` of size 10. Average time difference between the first and last touch event is recorded for different mesh sizes. Results show that the order of the punching does not affect the performance as the three lines are aligned for different mesh sizes. Overhead time varies between 1 millisecond up to $\approx$ 2 seconds for a 40x40 mesh size. This can be attributed to the accumulative waiting time in the `Threadpool` itself or the event queue. Thus, selective punching is recommended in this case as it will not require as many clicks to scan critical UI regions as whole screen punching would need. GenyMotion emulator [31] was used for testing with API 22. The emulator is set on 1 processor, 1024 MB Base Memory.

B. System level: Introducing OnOverlap Event

Currently, an activity can not know if its display is being covered by floating window. Thus, there is no action can be taken by the activity to either protect input fields or at least warn users of possible fraud. This approach requires modification of Android platform; thus, we have modified Android API 19 (Kitkat) as an example to enable detection of overlays. The modified version enables listening to a new event `OnOverlap` which will fire when a window of type SYSTEM_ALERT_WINDOW or TYPE_PHONE is overlapping an activity window. This enables developers to define custom behavior such as alerting the user or dimming the screen to prevent input of critical or sensitive data. The set of

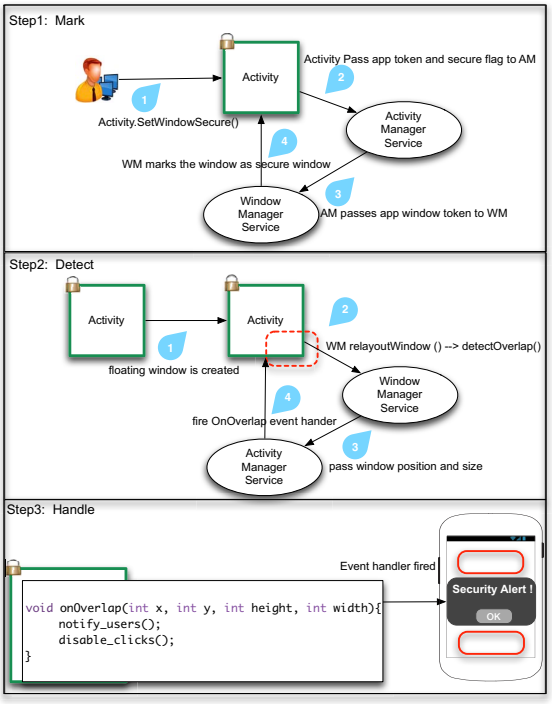


Fig. 11: Flow of handling floating windows in proposed framework

changes we have added can be categorized into three stages: Marking, Detecting and Handling. See Fig. 11.

Mark Secure Window: By default activities are not sensitive to overlays. We provide an approach for the developer to specify secure activities that react to overlays. Developer needs to call the new method `setSecureWindow` added to `Activity` class. `ActivityManager` service receives this change and passes it to `WindowManager` which now can recognize the window of the secure activity. Many classes have been modified to be able to represent and pass this flag including `Activity` class and `AppWindowToken` in `WindowManager`.

Detect Floating Window: It is essential to check for floating windows whenever a new window is added. In the class `WindowManager` the method `relayoutWindow` is called whenever a window is created, re-sized or redrawn. The function `detectOverlap()` is called only when two conditions hold: There are one or more application activity window drawn and there exists a window of type PHONE_TYPE or SYSTEM_WINDOW_TYPE. This function is to check if there is any floating window above a marked activity window. The algorithm used to detect overlapping windows

```
Result: Fire OnOverlap if detected
for All displayed windows on the screen do
    if secure window is found then
        for All windows on the top of secure window do
            Check window type;
            if TYPE_SYSTEM_ALERT or PHONE_TYPE
            is found then
                Fire OnOverlap event in the Activity
                displayed in secure window;
                Exit routine ;
            end
        end
    end
end
```

Algorithm 1: Detect Overlap algorithm

can be summarized in Algorithm 1. If there was a floating window detected the method sends a message to the event queue containing the position of the floating window and its height and width. Event queue, send that using `Handler` to Activity Manager service which identifies the corresponding app through `AppWindowToken`, Activity thread fires the OnOverlap event of the activity. At this stage, we have also added code to draw a red frame around the detected window to enable the user to see it.

Handle Floating Window: This simulates the `OnClick` event handler. The `OnOverlap` handler returns not only if there is an overlap with another window, it also returns information about the window size and location (see Fig. 12). This can be used by the developer to highlight the

```
void onOverlap(int x, int y, int height, int width){
     notify_users();
     disable_clicks();
}
```

Fig. 12: OnOverlap Event Handler

window borders, so that if floating windows are invisible , they method draws a line around them to visualize it to the user. Developers can also notify users by displaying an explicit message warning her not to enter her information. This will help ensure visual integrity and protect user data.

All these changes needed were done in one layer of Android; the Application Framework Layer in Android Architecture. We have modified 6 java classes related to the following components: Window Manager, Activity Manager and Activity adding extra ≈120 lines of code. Classes modified were: `WindowManager`, `ActivityManager`, `Activity`, `ActivityRecord`, `ActivityInfo`, and `ActivityThread`. The modified SDK can be accessed upon request. We have also created a patch file for the stock version of Android (Kitkat). Applying the patch file will modify stock version source code so that the new version reacts to overlays the way explained in this section.

Performance Evaluation We have recorded 700+ time stamps to measure execution time for `detectOverlap` having varying number of floating windows. Fig.13 shows the absolute execution time ranges from 1.48 ms for detecting one window to ≈ 3 ms needed to detect 10 floating windows, averaging of 2.7 ms. The routine `detectOverlap` is called in the function `relayoutWindow` that belongs to the class `WindowManager`. The call to the outer function is done when ever there is a need to redraw the screen , i.e. a window is added, removed or moved. So to measure how much overhead the function `detectOverlap` is adding to the original function, we have conducted several experiments. In each one there was a specific number of windows floating. We have monitored the time needed by the outer function (Δ_1) and the time needed by the inner function (Δ_2). The relative overhead added by the inner function is computed by: $\Delta_2 / \Delta_1 * 100\%$ which represents a percentage. Fig.14 shows that execution time of the function `OnOverlap()` was negligible compared to the outer function `relayoutWindow()` as the ratio of inner function to outer function averages 1.7%, ranging from 1.3% to 2.28%. In every experiment, approximately 700+

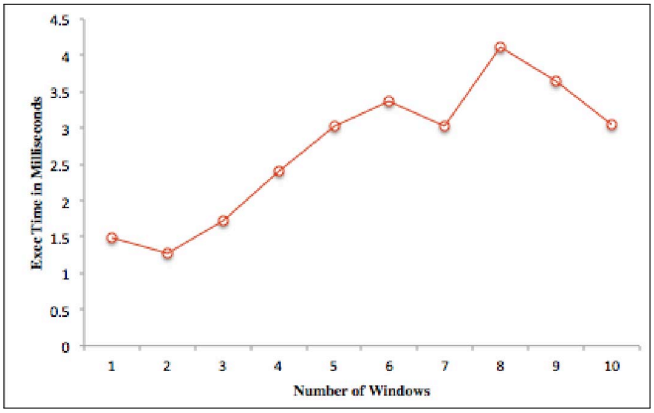


Fig. 13: detectOverlap Execution Time

time readings were recorded and averaged. The experiment was conducted on the built-in emulator of the modified version of Android Kitkat 4.4 . The emulator runs on memory of 512 MB, 200 MB internal storage and 64 MB VM Heap memory.

VI. Discussion & Conclusion

We have considered security implications of using floating windows on Android. While this feature is advertised as a "cool" feature, current Android implementation fails to support

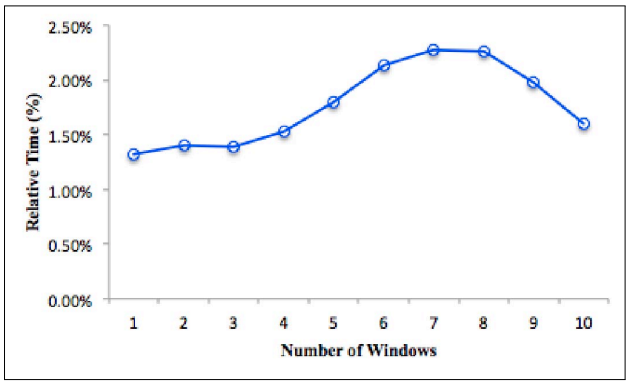


Fig. 14: Relative Execution Overhead

securing apps against malicious floating windows. We thus have modified Android OS to fire an event on the system level if floating window is detected. The resulting system enables developers to define custom behavior to deal with possible malicious floating windows. At this point, the proposed implementation will not be able to handle false positives,i.e. if the Activity flag is set, it will be sensitive to *any* floating window even if it was not a malicious one. That will be left to the developer to handle. Developers have the best judgment, wither to react to any floating window the same way through different app stages.

A simple application level solution is also suggested by introducing `SecureActivity` which can detect floating windows by injecting touch events. Time overhead of both approaches is measured.

Our research is a first step towards understanding the security risks associated with floating windows on Android, as we also believe that there is a gap in research related to Android display management in general and more specifically, its security implications. Future work should investigate how serious this issue is by scanning market apps that use floating windows. More work should be done in the direction of analyzing the purpose and the context of their usage.

VII. Acknowledgment

This work was supported in part by a Google Research Award. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Google. We would like to thank Dayabaran Gangatharan for his help in implementing the system based approach, as well as the reviewers of this paper.

References

[1] "Paranoid android," http://paranoidandroid.co.
[2] "Xposed module respository," http://repo.xposed.info/module/com.zst.xposed.halo.floatingwindow.
[3] "Standout," http://pingpongboss.github.io/StandOut/.
[4] "Google play- floating apps- multitaks," https://play.google.com/store/apps/details?id=com.lwi.android.flapps&hl=en.
[5] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson, "Busting frame busting: a study of clickjacking vulnerabilities at popular sites," *IEEE Oakland Web*, vol. 2, p. 6, 2010.
[6] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schecter, and C. Jackson, "Clickjacking: Attacks and defenses." in *USENIX Security Symposium*, 2012, pp. 413–428.
[7] H. Shahriar, V. K. Devendran, and H. Haddad, "Proclick: a framework for testing clickjacking attacks in web applications," in *Proceedings of the 6th International Conference on Security of Information and Networks*. ACM, 2013, pp. 144–151.
[8] M. Balduzzi, M. Egele, E. Kirda, D. Balzarotti, and C. Kruegel, "A solution for the automated detection of clickjacking attacks," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*. ACM, 2010, pp. 135–144.
[9] D. Akhawe, W. He, Z. Li, R. Moazzezi, and D. Song, "Clickjacking revisited a perceptual view of ui security," *BlackHat USA, August*, 2013.
[10] "Clickjacking rootkits for android: The next big threat?" https://news.ncsu.edu/2012/07/wms-jiang-clickjack/.
[11] M. Niemietz and J. Schwenk, "Ui redressing attacks on android devices," *Proceedings of BlackHat Abu Dhabi*, 2012.
[12] T. Luo, X. Jin, A. Ananthanarayanan, and W. Du, "Touchjacking attacks on web in android, ios, and windows phone," in *Foundations and Practice of Security*. Springer, 2013, pp. 227–243.
[13] E. Miluzzo, A. Varshavsky, S. Balakrishnan, and R. R. Choudhury, "Tapprints: your finger taps have fingerprints," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*. ACM, 2012, pp. 323–336.
[14] Z. Xu, K. Bai, and S. Zhu, "Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors," in *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*. ACM, 2012, pp. 113–124.
[15] "Clickjacking," http://www.sectheory.com/clickjacking.htm.
[16] U. Rehman, W. A. Khan, N. A. Saqib, and M. Kaleem, "On detection and prevention of clickjacking attack for osns," in *Frontiers of Information Technology (FIT), 2013 11th International Conference on*. IEEE, 2013, pp. 160–165.
[17] G. Rydstedt, B. Gourdin, E. Bursztein, and D. Boneh, "Framing attacks on smart phones and dumb routers: tap-jacking and geo-localization attacks," in *Proceedings of the 4th USENIX conference on Offensive technologies*. USENIX Association, 2010, pp. 1–8.
[18] "Look-10-007 tapjacking," https://blog.lookout.com/look-10-007-tapjacking/, 2010, accessed: 2015-04-1.
[19] 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. 23rd USENIX Security Symposium (SEC14), USENIX Association*, 2014.
[20] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 1036–1046.
[21] F. Roesner and T. Kohno, "Securing embedded user interfaces: Android and beyond." in *USENIX Security*, 2013, pp. 97–112.
[22] S. Fogie, J. Grossman, R. Hansen, A. Rager, and P. D. Petkov, *XSS Attacks: Cross Site Scripting Exploits and Defense*. Syngress, 2011.
[23] "Android: Activities," http://developer.android.com/guide/components/activities.html.
[24] "Android: Activity class," http://developer.android.com/reference/android/app/Activity.html.
[25] "What is windowmanager in android?" http://stackoverflow.com/questions/19846541/what-is-windowmanager-in-android.
[26] "Android design patterns," http://www.androiddesignpatterns.com/2013/07/binders-window-tokens.html.
[27] "An in-depth introduction to the android permission model and how to secure multi-component applications," https://www.owasp.org/images/c/ca/ASDC12-An_InDepth_Introduction_to_the_Android_Permissions_Modeland_How_to_Secure_MultiComponent_Applications.pdf.
[28] "Android binder," http://elinux.org/Android_Binder.
[29] "Instrumentation," http://developer.android.com/reference/android/app/Instrumentation.html.
[30] "Robotium," https://github.com/robotiumtech/robotium.
[31] "Genymotion," https://www.genymotion.com/#!/.