Touchjacking Attacks on Web in Android, iOS, and Windows Phone[⋆]

Tongbo Luo, Xing Jin, Ajai Ananthanarayanan, and Wenliang Du

Syracuse University, Syracuse NY, USA

Abstract. To make it easy for applications to interact with the Web, most mobile platforms, including Android, iOS, and Windows Phone, provide a mechanism that allows applications to embed a small but powerful browser component inside. This mechanism is called WebView in Android (it is called different names in other platforms). WebView implements a number of APIs that can be used by applications to interact with the web contents inside WebView. It has been pointed out by the previous work that malicious applications can use these APIs to attack the web contents inside WebView. Proposals are made by the previous work to fix the problems of those APIs. We have discovered that by fixing those APIs, WebView is still not secure. This is because the previous work only focuses on the APIs specifically designed for WebView; they have overlooked the APIs that WebView inherits from its super classes. These APIs are designed for the general-purposed user interface (UI) components, and they seem to pose no risk to those components; however, the combination of these APIs with the Web has led to new risks. We have identified several attacks based on these APIs. Our attacks are called Touchjacking attacks. They treat WebView as a blackbox, i.e., they do not use the APIs that are designed specifically for WebView; instead, they only use the inherited APIs. Through these APIs, malicious applications can attack the web contents inside WebView. The impact of the attacks is quite significant, as all the platforms that we have studied, including Android, iOS, and Windows Phone, are vulnerable to these attacks.

1 Introduction

In most mobile platforms, including Android, iOS, and Windows Phone, web browser is not just a stand-alone application anymore, it can be incorporated into applications. This is achieved by exposing web browser as a reusable component that can be embedded by applications. Such a component is called `WebView` in Android, `UIWebView` in iOS, and `WebBrowser` in Windows Phone. For the sake of simplicity, we only use the term WebView throughout this paper.

WebView makes it very convenient for developers to integrate browser functionalities, such as web page rendering, navigation, and JavaScript execution,

[⋆] The project was supported by the Google Research Award and the NSF Award No. 1017771.

into their mobile applications. Applications requiring basic browser functionalities can simply include the WebView library and create an instance of WebView class. The use of WebView is pervasive. In the Android Market, 86 percent of the top 20 most downloaded Android apps in each of the 10 categories use WebView.

WebView APIs. There are two types of APIs in WebView. One type is the APIs implemented by the classes associated with WebView. These APIs are designed for applications to interact with the web contents. We call this type of APIs the web-based APIs. Examples of these APIs include `loadURL`, `addJavascriptinterface`, `CookieManager.getCookie`, etc. The attacks described in [7] target the web-based APIs. The other type of APIs are those inherited. WebView is a specialized user interface (UI) component, and like others, such as buttons and text fields, it is designed as a subclass of the more generic UI components, such as the `View` class. As a result, WebView inherits its super classes' APIs. We call this type of APIs the UI-based APIs.

To enable interactions, WebView implements several APIs, allowing mobile application code (from outside WebView) to interact with the web contents, and JavaScript code (from inside WebView) to interact with the mobile application contents. Luo et al. pointed out that these APIs, if not properly protected, can lead to security problems [7]. Luo et al. has studied how those malicious apps launch attacks on the web contents inside WebView by taking advantage of lacking of access control on those web-based APIs and hooks [7]. However, once a better access control is enforced on the communication channel, the attacks can be defeated which is not difficult to achieve. For example, the `loadURL` is one of the most dangerous APIs used in Luo's attacks, but it turns out that most applications do not use this API to inject JavaScript code into WebView. Therefore, an easy solution is to modify this API and restrict it to load URL only, instead of allowing it to inject JavaScript code.

Assume such an access control system can be implemented in WebView, and all the vulnerable APIs of WebView are protected, the question is whether WebView is safe now. A complete access control by WebView should control all the potential interaction channels between applications and WebView. No study has looked at whether the UI-based APIs inherited by WebView can pose risks to the contents that reside inside the webView. The objective of this work is to study the feasibility of attacks using the UI-based APIs. As Figure 1(a) illustrated, our attacks will not use any of the web-based APIs designed for WebView. In other words, even if the problems described in [7] are fixed, there are still ways to attack the contents in the blackbox WebView.

As we all know, when software components are reused (e.g., through libraries or class inheritance), their features, although safe and appealing for other systems, may bring danger to new systems. For WebView, it was not clear whether those inherited UI-based APIs pose any threat in the new systems, especially whether they can be used by malicious applications to attack the contents within WebView. There has been no study to investigate the security impact of those UI-based APIs inherited by WebView, mostly because these UI-based APIs have not appeared to be problematic to other UI components. After studying Web-

View, we realized that the attacks conducted by Luo et al. only covered one type of interaction, not all.

Security Concerns on UI-based APIs. From the security perspective, there is one thing that clearly separates WebView from the other UI components, such as buttons, text field, etc. In those UI components, the contents within the components are usually owned by or are intended for the applications themselves. For example, the content of a button is its label, which is usually set by applications; the content of a text field is usually user inputs, which are fed into applications. Therefore, there is no real incentive for applications to attack the contents of these components. WebView has changed the above picture.

In mobile systems, the developers of applications and the owners of web contents inside WebView are usually not the same. Contents in WebView come from web servers, which are usually owned by those that differ from those who developed the mobile applications. It should be noted that before Facebook released its own applications for iPhones and Android phones, most users used the applications developed by third parties (many are still using them). For example, one of the most popular Facebook apps for Android is called FriendCaster for Facebook, which is developed by Handmark, not Facebook. Because of such an ownership difference, it is essential for all mobile platforms to provide the assurance to web applications that their security will not be compromised if they are loaded into another party's mobile applications.

A WebView component with better access control enforced on all the cross-component communication channels can be treated as a blackbox. The mobile system guaranteed that the integrity and confidentiality of the web applications cannot be compromised even if they were loaded into the WebView embedded in a malicious application. Although users may not fully trust the third-party mobile apps, they fully trust the system once they make sure that they are using the WebView. The similar trust assumption is made when users view private contents in an iframe which is embedded in a third-party mashup web application. This is because users trust the isolation mechanism enforced by the browser to constraint the access from the host webpage if it comes from a different domain.

Overview of our work and contribution. In this paper, we systematically studied the security impact of these UI-based APIs. Our attack model is the following: First, we assume that the mobile application is malicious; it embeds one or more WebView components in it. The target of the attack is the web contents within the WebView components. The attackers are interested in stealing sensitive information from the web page, or compromising the integrity of the web page and its interaction with user.

We further assume that the attackers can only use the UI-based APIs inherited by the WebView class. In other words, the malicious applications cannot directly interact with the web contents inside WebView. This assumption will significantly distinguish our work from that by Luo et al. [7], which focuses only on the web-based APIs. Putting this assumption in a different way, we are investigating whether WebView can be secured if it is redesigned to address the attacks in [7].

We have identified several different attacks that can be launched on WebView solely using the UI-based APIs. We have studied these attacks in three popular mobile phone platforms, including Android, iOS, and Windows Phone. All of them are vulnerable to our attacks. For the sake of simplicity, we will only talk about WebView and the attacks in the context of Android.

Our discoveries are significant, as they demonstrate that securing and redesigning the web-based APIs are not sufficient; we also need to study the APIs that WebView inherited from its super classes, understand how dangerous they are to the web contents inside the WebView, and find solutions to secure them. This paper only focuses on the attack part; developing solutions to solve the problem for Android, iOS, and Windows Phone is still a work-in-progress, and will be published in our future papers.

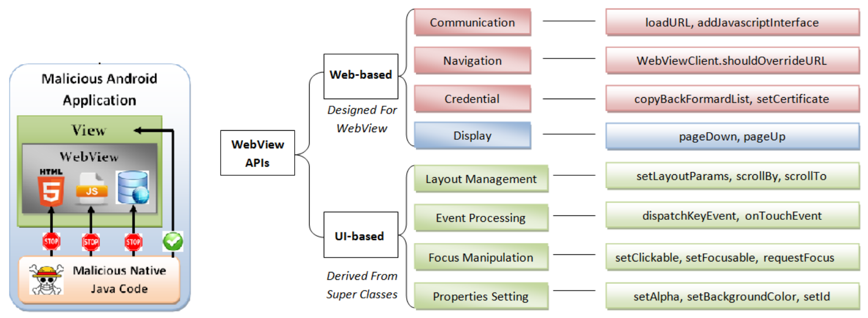
2 WebView APIs

To enable applications to browse the Web from within themselves, instead of using an external browser application, Android provides a package called `android.webkit` to applications. This package contains several classes, each for different purposes. The most important class among them is called `WebView`, which is a View class that displays web pages. This class is the basis for displaying web contents within applications. It uses the WebKit rendering engine to display web pages; it also includes methods to navigate forward and backward, zoom in and out, perform text searches, etc. In addition to WebView, `android.webkit` also includes several other classes related to the Web, such as `CookieManager` (for managing cookies), `WebViewClient` (for customized handling of events within WebView), `WebViewDatabase` (for managing the WebView database), etc.

Jointly, these classes expose many APIs to Android applications. Based on their purposes, these APIs can be divided into two main categories (see Figure 1(b)). One category is the APIs that are designed for the control of web pages and their related data (e.g. cookies, histories, and caches), and we call them the web-based APIs. The other category is the APIs that are derived from their super classes, which are designed for the general user interface (UI) components, and we call them the UI-based APIs.

2.1 Web-based APIs

The classes in the `android.webkit` package jointly expose a number of APIs to the applications for better manipulation and control over the web contents inside WebView. Those APIs are quite useful for application developers to embed and customize "browser-like" components within applications, and thus enrich the functionalities of applications. We will not go over all those APIs; we only describe those that are related to security.


Malicious Android Application
View
WebView
HTML
JS
Malicious Native Java Code
WebView APIs
Web-based
Designed For WebView
UI-based
Derived From Super Classes
Communication
Navigation
Credential
Display
Layout Management
Event Processing
Focus Manipulation
Properties Setting
loadURL, addJavascriptInterface
WebViewClient.shouldOverrideURL
copyBackFormardList, setCertificate
pageDown, pageUp
setLayoutParams, scrollBy, scrollTo
dispatchKeyEvent, onTouchEvent
setClickable, setFocusable, requestFocus
setAlpha, setBackgroundColor, setId

(a) Threat Model (b) WebView APIs Overview

Fig. 1. WebView APIs

Webpage-Android Communication. Android WebView provides a bidirectional communication channel between the webpage environment inside WebView and the native Android application runtime. For example, WebView provides a mechanism for the JavaScript code inside it to invoke Android apps' Java code. The API used for this purpose is called `addJavascriptInterface`. In addition to the JavaScript-to-Java interaction, WebView also supports the interaction in the opposite direction, from Java to JavaScript. This is achieved via another WebView API `loadUrl`.

Webpage-related Hooks. Android applications can monitor the webpage navigation and rendering events occurred inside WebView. This is done through the hooks provided by the `WebViewClient` class. These hooks will be triggered when their intended events occur inside WebView. Once triggered, these hooks can access the event information, and may change the consequence of the events. For example, by overloading the hook `shouldOverrideURL`, Android applications can intercept and modify the destination URL when the user tries to navigate to another web page or site.

Webpage Credentials. All the credentials and private data of web pages are stored in an internal database, which is isolated from Android applications. However, WebView exposes many APIs to allow applications to fetch or modify the sensitive webpage contents in the internal database. For example, Android applications can directly inject the certificate of a webpage through the API `setCertificate`, cookies can be accessed using `CookieManager.setCookie`, and so on.

2.2 UI-based APIs

The `android.webkit` package includes a number of classes, most of which inherit directly from `java.lang.Object`, which is the root of all classes in Java. The APIs inherited from this root class do not pose much risk. An outlier among these classes is the `WebView` class, which is the main UI class in the package. This class inherits the APIs from several classes. Its inheritance tree is the following (starting from the root): `java.lang.Object`, `android.view.View`, `android.view.ViewGroup`, and `android.widget.AbsoluteLayout`. Moreover, `WebView` also implements seven interfaces, with six of them coming from the `android.view` package, and one from `android.graphics` [1].

Among all the classes and interfaces inherited by `WebView`, the most significant class is `Android.view.View`, which is commonly used by Android applications. The View class represents the basic building block for user interface components; it usually occupies a rectangular area on the screen and is responsible for drawing and event handling. This class serves as the base for subclasses called `widgets`, which offers fully implemented UI objects, like text fields and buttons. `WebView` is just a customized widget.

Our attacks focus on the APIs provided by `Android.view.View`. These APIs can be classified into several categories, including Layout Management, Event Processing, Focus Manipulation, and Properties Setting, all of which are the basic functionalities designed for native Android UI objects. We will illustrate some of the commonly used APIs in this `View` class. It should be noted that some of the APIs inherited from the `View` class are overridden in the `WebView` class, but we still count them as the UI-based APIs.

Layout Management. One of the basic features of Android UI objects is to provide basic methods to handle the screen layout management. For example, a view object has a location (expressed as a pair of left and top coordinates) and two dimensions (expressed as a width and a height). Android applications can use the methods, such as `layout`, `setX`, and `setMinimumHeight`, to configure locations. It is possible to retrieve the location of a view object by invoking the methods `getLeft` and `getTop`. Similar methods can also be used to get the size information of the WebView.

Android also provides basic supports for views that need scrollbars. This includes keeping track of the X and Y scroll offsets as well as drawing scrollbars. Using the methods like `scrollBy` and `scrollTo`, Android applications can control the displayed area of the content in the view object. Obviously, for WebView, the contents inside the WebView are web pages.

Event Processing. Each Android view object is responsible for drawing the rectangular area on the screen that it occupies, and handling the events in the area. Views allow clients to set listeners through hooks that will be notified when something interesting happens to the view. For example, by using the method `setOnKeyListener`, Android applications can register an event handler callback function which will be invoked when a key is pressed in this view. Besides intercepting the events, the view class also exposes methods for Android applications to pass motion events down to the target view.

Focus Manipulating. The Android framework will handle moving focus in response to user input. To force focusing on a specific view, applications can call `requestFocus()` of that view.

Properties Setting. Other advanced features related to appearance could be the background color or alpha property of WebView, like methods `setBackgroundColor` and `setAlpha`.

2.3 Attack Model

For all of the attacks described in this paper, we have the following assumptions:

1. **We are concerned about potential malicious applications in mobile devices.** As we pointed out, the developers of the apps and the owner of the web contents inside WebView are usually not the same. Our investigation shows that among the top 113 apps that use WebView, 49 are third-party apps. Therefore, it is quite common for web contents to be loaded into an untrusted environment.
2. **We assume that the users clearly know they are using WebView.** Users make sure they are using the secured blackbox WebView instance to access web contents, and they trust that the mobile system can isolate the contents inside WebView from those from outside.
3. **We assume that the effective access control mechanism is already enforced on the Web-based APIs exposed by the WebView.** As we mentioned before, Web-based APIs are powerful to control the web contents inside WebView. We assume a perfect redesigned access control model has been implemented on WebView to isolate the contents inside WebView from outside world. This assumption clearly distinguish this work from that in [7], because under such an assumption, the attacks describe in [7] will not be threats any more.
4. **We assume that the UI-based APIs are accessible by the apps.** WebView is a specialized user interface (UI) component, and like others, such as buttons and text fields, it is designed as a subclass of the more generic UI components, such as the `View` class.
5. **We assume that malicious apps are only granted with one permission.** It should be noted that to successfully launch the attacks described in this paper, malicious Android applications only need one permission `Android.permission.INTERNET`. This permission is widely granted to 86.6% of free (and 65% of paid) Android applications [4]. Generally speaking, these attacks are relatively easy to launch and difficult to detect, since they only require one very common and less-dangerous permission.

3 Touchjacking Attacks

In this section, we describe how to let users generate touch events, and how to hijack those events for malicious purposes. We call this type of attacks the `Touchjacking attack`. We describe three attacks; based on their different attack strategies, we give them different names.

We give a brief overview of the three attacks here, and explain the details later in this section. Figure 2 illustrates the attacks.

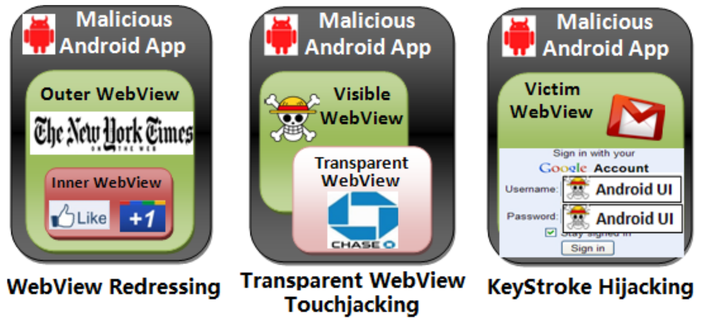


Fig. 2. Touchjacking Attack Overview

1. **WebView Redressing Attack.** In this attack, malicious applications put a smaller WebView on top of a larger one, making the smaller one look like an element (e.g. button) within the larger one.
2. **Invisible WebView Attack.** In this attack, malicious applications overlay an invisible WebView on top of a visible one, causing users to see the visible one, but operate on the invisible one.
3. **Keystrokejacking Attack.** In this attack, malicious applications overlay some native UI objects on the top of the HTML elements inside WebView; while the user believe that they are typing in the field that belongs to a web page, they are actually typing in a field that belongs to the malicious applications, which can steal the information typed by the users.

3.1 Positioning Method

By default, after being loaded into a WebView, the webpage will be displayed inside the WebView. If the size of the webpage is larger than the size of the WebView, only the most top-left area of the webpage will be displayed initially. However, in order to carry out our attacks, certain HTML elements (e.g. a button) of the targeted webpage must be carefully positioned. Only using the traditional positioning methods that facilitate clickjacking attack in browsers is not enough to meet the positioning requirement for Touchjacking attacks. We describe some positioning techniques.

Pixel Coordination. Android applications can use the following APIs to position a web page to a specific position inside WebView: `scrollBy`, `scrollTo`, `pageDown`, `pageUp`. The method `scrollBy(x,y)` scrolls the page by x pixels horizontally and y pixels vertically; the method `scrollTo(x,y)` scrolls the page to the `(x, y)` position. The method `pageDown` and `pageUp` scroll the display area to the top and bottom of a webpage. Attackers can also use the websetting APIs to change the font size or zoom level of the webpage, such as `setTextSize` and `setDefaultZoom`.

URL Fragment Identifier. Using pixel coordinates to position a target can be inaccurate due to other factors, such as rendering differences between browsers and font size differences between platforms. A solution to this problem is to use the URL fragment identifiers to position anchor elements of the webpage. Anchors and URL fragments are commonly used together to link to a particular section of the text within an HTML document. When a URL containing a fragment identifier is loaded, a browser will scroll the page so that the anchor is at the top of the viewable area. An anchor can be created in two ways, either by adding a `'name'` attribute to an `'a'` tag, or by adding an `'id'` attribute to any element. The following example shows how to navigate to the specific `div` tag using URL fragment identifiers.

3.2 WebView Redressing Attack

Generally speaking, the idea behind the WebView redressing attack is to seamlessly merge two or more WebView containers, making them look like one. When the non-suspicious user reacts to the contents inside WebView by clicking some links or buttons, because what the user clicks on may belong to a different page in another WebView, the user is tricked into reacting to the contents in another WebView, and those contents are not even displayed to the user.

The attack consists of two or more WebViews (we will use two in our description). One of the WebViews is called the outer WebView, and the other is called the inner WebView. The inner WebView loads the malicious webpage, and it is intentionally made small, so it only displays a very small portion of the webpage to users. This is important, as the attackers do not want the users to see the entire page, which reveals the malicious intents. The malicious application can use the positioning method described above to display a specific part of the page (such as a button) to users.

The outer WebView is larger, and is for the users to view web contents. Attackers overlay the inner WebView on top of the outer WebView, and make it cover a selected area of the outer WebView. Because the inner WebView is small and has no obvious boundaries, the inner WebView looks like part of the elements on the webpage inside the outer WebView. If users react to the contents in the outer WebView, and clicks on the buttons within the inner WebView, they are actually reacting to the contents in the inner WebView. This is dangerous, as the users never got a chance to see the contents that they have reacted upon.

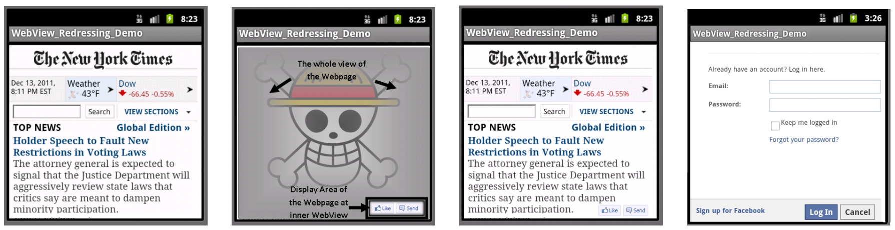
Case Study. We demonstrate the WebView redressing attack using an example. Facebook has been a major spam target; one of the goals of the spammers is to find ways to post links or other information on Facebook user's walls. Just like email spams, no matter what improvement the company makes, spammers have always been able to find new ways to cause problems. We will demonstrate a new way to launch the "likejacking attack" [13] by using the WebView redressing technique, so that the users can be tricked into "Like"ing spam pages.

In this attack scenario, assume that the malicious Android application is written for New York Times. Normally, only the outer WebView is visible and users will use this WebView to visit the articles at www.nytimes.com (see Figure 3(a)). The malicious Android application can insert the inner WebView at any time when the user navigates to the New York Times page. The inner WebView contains the spam article, with a Like button (see Figure 3(b); we did not show the spam article in the figure). The attackers need to pre-calculate the location of the inner WebView (Figure 3(b)) to redress the webpage inside the outer WebView.

After the redressing, what the user sees is shown in Figure 3(c). Clearly, it is quite difficult for the user to see that the Like button is not part of the New York Times page. If the user really likes this article and wants to share it through Facebook, he/she will click on the Like button, not knowing that the button is associated with a different article hidden in another WebView.

If the user has not logged into Facebook yet from this application, once clicking on the Like button of the inner WebView, a dialogue window (which is a new WebView instance) will be popped up with the Facebook's login page inside (see Figure 3(d)). Since it is hard for the user to realize that the dialog window is not popped up by the outer WebView, the user may very likely log into Facebook, and eventually share the article that he/she has never seen.

It is also likely that the user may have already logged into Facebook from the inner WebView (due to the clicking of some legitimate Like buttons). Because cookies are shared among all the WebView instances within the same Android application, clicking on the Like buttons in another WebView will not result in the pop-up dialogue window; instead, the "like" request will be automatically sent to Facebook with the valid cookies.



Holder Speech to Fault New Restrictions in Voting Laws
The attorney general is expected to signal that the Justice Department will aggressively review state laws that critics say are meant to dampen minority participation.

Already have an account? Log in here.

(a) Outer WebView (b) Inner WebView (c) What User Sees (d) Login Dialog

Fig. 3. WebView Redressing Attack Example

3.3 Invisible WebView Attack

Both Android and iOS systems allow applications to set transparency on WebView (UIWebView) objects. Low opacity may result in the webpage inside WebView being hardly visible, or completely invisible. In Android 3.0, applications can use the method `setAlpha` to set the opaque level of the WebView object. Every native Android UI object maintains the alpha property and exposes the setter and getter to applications. Since the `WebView` class was derived from the `View` class, it also inherits this property. It should be noted, when a WebView object is transparent (i.e. alpha value equals to 0), it is transparent visually, not physically, i.e., users can still touch/click on the page inside a transparent WebView. The following code demonstrates how to set the WebView transparent.

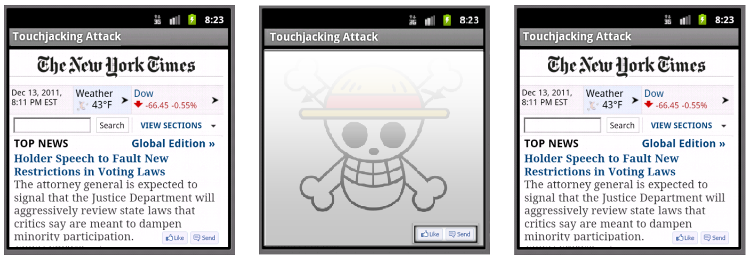
```
WebView mWebView = (WebView) findViewById(R.id.webview);
mWebView.setAlpha(0);
```

The transparency feature is intended for generic UI components, and it brings no harm to them; however, when this feature is inherited by WebView, it poses great danger to the web contents inside WebView. We describe how this feature can be used for attacks.

In this attack, malicious Android applications need to have two WebView instances: one visible and the other invisible. The `visible` WebView will load an attractive webpage that is controlled by attackers, and the purpose of this page is to entice users to perform touch actions. For example, this web page can be a small game. Another WebView is `invisible`, and it loads the targeted webpage. The invisible WebView is put on top of the visible one. Therefore, when the user touch something that is apparently in the visible WebView, the touch actually goes to the invisible one, because it is on the top.

To successfully launch the touchjacking attack, attackers need to first calculate the position where user may perform the touch action. Since the attacker controls the visible webpage, it is not hard to predict the position and precisely overlay the UI in the targeted webpage inside the invisible WebView object on top of specific position. Attackers can use the positioning techniques mentioned in the beginning of this section to control the place of the clickable elements (e.g. button, link).

Case Study 1. In this attack example, we repeat the case study in the previous subsection, but using the transparency technique to achieve the same goal. We assume that the malicious Android application is written for New York Times, and the user is currently reading an article from there. This time, the article itself has a legitimate Like button to facilitate sharing via Facebook (see Figure 4(a)).



Holder Speech to Fault New Restrictions in Voting Laws

The attorney general is expected to signal that the Justice Department will aggressively review state laws that critics say are meant to dampen minority participation.

(a) Visible WebView (b) Invisible WebView (c) User's View

Fig. 4. Invisible WebView Attack Example

Attackers create another WebView (invisible), and load the spam page inside it. This page contains an article that the spammers want the user to share with their Facebook friends, and there is a Like button on this page (Figure 4(b) shows this spam page, but we did not show the spam article inside).

The malicious application then overlays the invisible WebView on top of the visible one. Using the positioning techniques, the attackers can make the two Like buttons in both WebViews be placed at exactly the same location on the screen, i.e., they completely overlap. Because of the transparency, what the user sees is exactly the same as that in Figure 4(a).

When the user clicks on the Like button, the click event goes to whatever is on the top, i.e., the transparent WebView, not the one for New York Times. As results, the spam article is shared to the user's Facebook friends. This consequence is the same as that in the WebView redressing attack.

Case Study 2. If the user also uses the malicious application to log into his/her online accounts (such as Facebook), the attack can be much more severe. We use Facebook to demonstrate how to use the Invisible WebView attack to hijack the touch events and trick users into deleting friends from their Facebook accounts.

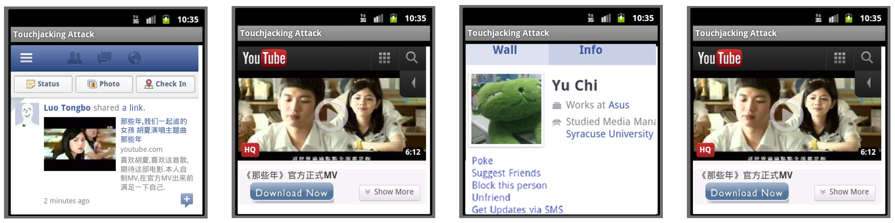
Before the attack is launched, users have logged into their Facebook accounts from the visible WebView, and are viewing their Facebook pages (Figure 5(a)). At this time, the invisible WebView is not overlaid yet. When the user clicks a link shared by his/her friend, WebView will navigate to another webpage (Figure 5(b)); this webpage is not malicious, but the attacker needs to know the possible click points. At the same time, the application needs to overlay the invisible WebView on the top, and inside the WebView should be the Facebook webpage (Figure 5(c)).

Attacker can also precisely put the `UNFRIEND` link of the transparent Facebook page on the top of the `DOWNLOAD` button of the visible WebView. If the user wants to download the video as shown in Figure 5(d), the user needs to click the `DOWNLOAD` button. Because the `UNFRIEND` button is on the top, this button is actually clicked, and user's some friends will be deleted from the friend list.

Although the user has never actually logged into Facebook account using the invisible WebView, since the cookies are shared among all WebView instances within the same Android application, the UNFRIEND request from the webpage in the invisible WebView will be able to attach the Facebook cookies and cause the deletion of the user's friend.

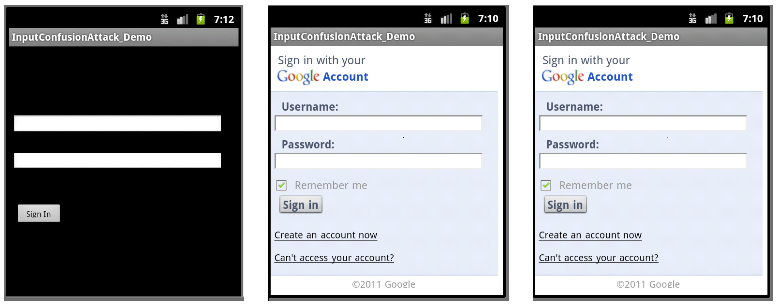
3.4 Keystroke Hijacking Attack

In the previous attacks, attackers redirect the user's actions toward the webpage in a WebView instance that is different from what the user sees. In this attack, we will demonstrate how attackers can redirect those actions to the native Android UI objects (e.g. a text field) that is completely controlled by the malicious applications. If the user's actions involve secrets (e.g. passwords), the attacker can get the secrets.



(a) Initial Visible WebView (b) Visible WebView (c) Invisible WebView (d) What User Sees

Fig. 5. Invisible WebView Attack Example


InputConfusionAttack_Demo
Sign In
Sign in with your Google Account
Username:
Password:
Remember me
Sign in
Create an account now
Can't access your account?
©2011 Google

(a) Native UI Only (b) WebView Only (c) User's View

Fig. 6. Keystroke Hijacking Attack Example

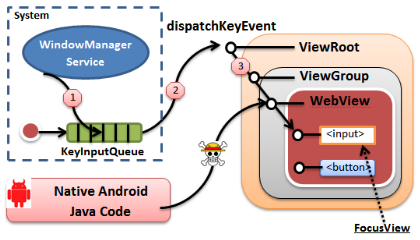
The attack is based on the fact that the HTML UI objects inside WebView and the Android native UI objects are based on the same GDI (`skia`), and the exterior appearance of the HTML UI objects look similar to their related native UI objects. For example, the HTML input field looks almost the same as the text editing widget `EditText`, which is a native UI component of Android. Therefore, if we put a native UI object on top of the HTML UI object of the same type, users will not be able to tell the difference. If they decide to type into what appears to be a part of the webpage, they will be typing into the native UI object that belongs to the attackers.

To successfully launch the attack, the attackers should precisely overlay the native Android UI objects on top of the HTML objects of the web page inside WebView, with exactly the same size and location. Since the layout of the victim page is almost stable in many cases (e.g. login pages), attackers can quite easily calculate the size and position of the targeted UI objects within the webpage.

Case Study. We use Gmail as an example to demonstrate how the attack works. We separately display the two layers of layout in the malicious applications. Figure 6(a) is the upper layer, consisting of two `EditText` native UI components. Figure 6(b) is the lower layer, consisting a WebView with the Gmail login page inside. When being displayed on the screen, the two `EditText` UI

API Name	**Event Type**	**Action**
dispatchDragEvent	DragEvent	Drag
dispatchKeyShortcutEvent	KeyEvent	Key
dispatchKeyEventPreIme	KeyEvent	Key
dispatchHoverEvent	MotionEvent	Touch
dispatchTrackballEvent	MotionEvent	Trackball
dispatchKeyEvent	KeyEvent	Key
dispatchTouchEvent	MotionEvent	Touch

(a) Event Dispatching APIs



(b) Event Dispatching Mechanism

Fig. 7. Event Dispatching Mechanism and APIs

components will exactly overlap with the two input fields on the Gmail login page. When users type the username and password, they actually type in the `EditText` UI components, which are accessible by the attacker.

Users may be aware of the attack once they finish the input actions and submit the form, because the actual HTML input objects are empty, and an error message will be displayed. To further disguise that, the attackers should also add a fake submit button (native UI object). Once the fake button is clicked, the malicious application should ask WebView to navigate to an error page, displaying something like "Page cannot be displayed due to network problems". After the users go back to the previous page, the malicious application remove all the overlaid native UI objects, so the users can proceed without raising suspicions.

4 Event-Simulating Attacks

The touchjacking attacks described earlier hijack real user's touch events, while this attack can generate fake touch events. As we have discussed in Section 2, like all of the view-based Android UI objects, the WebView class inherits a number of methods from the View class, including the ones needed by the event-dispatching mechanism in Android. Those APIs are listed in Figure 7(a). However, those event-dispatching APIs also have to be exposed to Android applications. As results, by invoking those APIs to dispatch the action to the currently focused HTML UI objects, applications can generate keystroke, click, and touch-screen events within WebView without consents from users (Figure 7(b)).

Since we believe that those APIs directly interact with the web page inside the WebView, attacks using those APIs will not treat the WebView as the blackbox. Although this attack is more powerful than the touchjacking attacks, we believe in the future, those APIs will be blocked by the android system. Due to the page limitation, we cannot cover the details of this attack here.

5 Attacks on Other Platforms

To see whether the attacks we identified in this paper work on the platforms other than Android, we have tried the attacks on iOS (version 4.3.2) and Windows

Phone 7. All the three types of Touchjacking attacks work on iOS and Windows Phone 7. For the event-simulating attack, unlike Android, iOS does not provide APIs to dispatch key/touch events to UIWebView. Therefore, we were not able to directly simulate key/touch events in UIWebView. Similar to iOS, Windows Phone does not provide any API support for programmatically invoking an event.

6 Related Work

Since the first bug report on the negative usage of iframe by [10], Clickjacking attack [6], UI redressing attack [8], and Next Generation Clickjacking attack [14] have been developed by taking advantage of transparent iframes. To successfully launch a clickjacking attack, a malicious page is constructed by attackers in a way that tricks victims into clicking on the elements in a different page that is only barely visible or completely invisible [15]. The project [12] introduced the `Tapjacking` attack, which takes advantage of the unique feature of mobile browsers to launch stronger clickjacking attacks on mobile devices. Tapjacking also uses iframes as the tool to launch the attack.

The Touchjacking attack described in our paper achieves the same goal as clickjacking attacks, but our work is the first to study the attacks using WebViews, instead of iframes. Because the tools used in the attacks are different, our attacks cannot be prevented by the solutions proposed for clickjacking attacks, such as frame busting [11] and X-Option-Header [5]. Like the clickjacking attack, the Touchjacking attack can also be considered as an instance of the confused deputy problem [2].

Touchjacking attacks differ from phishing [9] attacks because they do not trick users to enter secret credentials into a spoofed website. Instead, users need to enter their credentials into the real website in the WebView to establish an authenticated session. The attack can proceed until the user's session expires. The user's sensitive information is completely isolated from the malicious Android application throughout the attacks.

The project [3] discovered several phishing attacks that can be mounted against control transfer. Those attacks will take advantage of the lack of secure application identity indicators in mobile operating systems and browsers, so the user cannot always identify whether a link has taken him/her to the expected application. Some of the attacks target the phishing attack on browsers and WebViews, but they depend on faking the whole browser or using the WebView APIs to directly compromise the webpage. The spoofing attack we introduced in this paper is not due to any of the four control transfer scenarios, but a new control transfer from webpage to system.

Our paper is distinguished from the work [7], which tries to exploit WebView vulnerabilities by directly manipulating the contents inside WebView through the powerful APIs and hooks exposed by WebView. In this paper, we assume that all the access paths to directly communicate with the webpages inside WebView have been blocked or securely controlled.

7 Conclusion

The security problem of the WebView technology has been studied before, but the existing work focuses on how the APIs designed specifically for WebView can be abused to compromise the security of the web contents inside WebView. The work calls for adding extra access control into those APIs. This paper points out that even if those APIs are secured, WebView is still dangerous. This is because WebView inherits many UI-based APIs from its super classes, and those APIs can be abused as well, although in a very different way. We describe several attacks based on these APIs. We show that using these APIs, attackers can compromise the integrity and confidentiality of the web contents inside WebView.

References

1. Android-Team. Webview class reference. `http://developer.android.com/reference/android/webkit/WebView.html`.
2. T. Close. The confused deputy rides again! `http://waterken.sourceforge.net/clickjacking/`, 2008.
3. A. Felt and D. Wagner. Phishing on mobile devices. In *Web 2.0 Security and Privacy*, 2011.
4. A. P. Felt, K. Greenwood, and D. Wagner. The effectiveness of application permissions. In *Proceedings of the 2nd USENIX conference on Web application development*, WebApps'11, pages 7–7, Berkeley, CA, USA, 2011.
5. Firefox. The x-frame-options response header. `https://developer.mozilla.org/en/The_X-FRAME-OPTIONS_response_header`.
6. R. Hansen. Clickjacking. `http://ha.ckers.org/blog/20080915/clickjacking/`.
7. T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin. Attacks on webview in the android system. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 343–352. ACM, 2011.
8. M. Niemietz. Ui redressing: Attacks and countermeasures revisited. In *CONFidence 2011*, 2011.
9. Y. Niu, F. Hsu, and H. Chen. iphish: phishing vulnerabilities on consumer electronics. In *Proceedings of the 1st Conference on Usability, Psychology, and Security*, pages 10:1–10:8, Berkeley, CA, USA, 2008. USENIX Association.
10. J. Ruderman. Bug 154957 - iframe content background defaults to transparent. `https://bugzilla.mozilla.org/show_bug.cgi?id=154957,`, 2002.
11. G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. In *IEEE Oakland Web 2.0 Security and Privacy*, 2010.
12. 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*, pages 1–8. USENIX Association, 2010.
13. Sophos. Facebook worm - likejacking. `http://nakedsecurity.sophos.com/2010/05/31/facebook-likejacking-worm/`, 2010.
14. P. Stone. Next generation clickjacking, 2010.
15. M. Zalewski. Browser security handbook. `http://code.google.com/p/browsersec/wiki/Part2`, 2008.