Attacks on WebView in the Android System*

Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin
Dept. of Electrical Engineering & Computer Science, Syracuse University
Syracuse, New York, USA

ABSTRACT

WebView is an essential component in both Android and iOS platforms, enabling smartphone and tablet apps to embed a simple but powerful browser inside them. To achieve a better interaction between apps and their embedded "browsers", WebView provides a number of APIs, allowing code in apps to invoke and be invoked by the JavaScript code within the web pages, intercept their events, and modify those events. Using these features, apps can become customized "browsers" for their intended web applications. Currently, in the Android market, 86 percent of the top 20 most downloaded apps in 10 diverse categories use WebView.

The design of WebView changes the landscape of the Web, especially from the security perspective. Two essential pieces of the Web's security infrastructure are weakened if WebView and its APIs are used: the Trusted Computing Base (TCB) at the client side, and the sandbox protection implemented by browsers. As results, many attacks can be launched either against apps or by them. The objective of this paper is to present these attacks, analyze their fundamental causes, and discuss potential solutions.

1. INTRODUCTION

Over the past two years, led by Apple and Google, the smartphone and tablet industry has seen tremendous growth. Currently, Apple's iOS and Google's Android platforms take 64 percent of the market share, with Android taking 37 percent and iOS 27 percent [8]. Because of the appealing features of these mobile devices, more and more people now own either a smartphone, a tablet, or both. A recent Nielsen survey showed that nearly one third of US mobile users had smartphones at the end of 2010 [8].

A critical factor that has contributed to the wide-spread adoption of smartphones and tablets is their software applications (simply referred to as *apps* by the industry). These apps provide many innovative applications of mobile devices. There are many apps on the market; combined, iOS and Android have over 500,000 apps, for both smartphones and tablets, and the number is still increasing at a fast rate.

Among these apps, many are web-based. Namely, they get contents from web servers using the standard HTTP protocol, display the web contents, and allow users to interact with the web servers. It seems that they are doing exactly what can already be done by real browsers, but there are significant differences. Browsers are designed to be generic, and their features are independent from web applications. Most web-based apps, on the contrary, are customized for specific web applications. Because they primarily serve their intended web applications, they can implement features that are specific to those applications.

For example, `Facebook Mobile` is developed specifically for `Facebook` to provide an easier and better way—compared to `Facebook`'s web interface—to view `Facebook` content, interact with its servers, and communicate with friends. Because of the richer experience gained from these customized "browsers", most users prefer to use them on mobile devices, instead of the actual browsers. Many popular web applications have their dedicated apps, developed in-house or by third parties.

What enables apps to be customized for specific web applications is a technology called *WebView*, adopted by both Android and iOS (it is called UIWebView in iOS, but for simplicity, we simply use WebView throughout this paper). The WebView technology packages the basic functionalities of browsers—such as page rendering, navigation, JavaScript execution—into a class. Apps requiring these basic browser functionalities can simply include the WebView library and create an instance of WebView class. By doing so, apps essentially embed a basic browser in them, and can thus use it to display web contents or interact with web applications. 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.

What truly makes customization possible is the APIs provided by WebView. WebView not only allows apps to display web content, more importantly, through its APIs, it enables apps to interact with the web content. The interaction is two-way: From apps to web pages, apps can invoke JavaScript code within web pages or insert their own JavaScript code into web pages; apps can also monitor and intercept the events occurred within web pages, and respond to them. From web pages to apps, apps can register interfaces to WebView, so JavaScript code in the embedded web pages can invoke these interfaces.

*This work was supported by Award No. 1017771 from the US National Science Foundation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ACSAC '11 Dec. 5-9, 2011, Orlando, Florida USA
Copyright 2011 ACM 978-1-4503-0672-0/11/12 ...$10.00.

With such a two-way interaction mechanism between apps and web pages, apps become more powerful than the traditional browsers. They can customize their interfaces based on the web contents and the screen size, as well as provide additional features beyond what is provided by the web application, giving users a much richer experience than using the generic browsers. For example, `Facebook mobile` makes it easy to stay connected and share with friends, share status updates from the home screen, chat with friends, look at friends' walls and user information, check in to places to get deals, upload photos, share links, check messages, and watch videos. These features, implemented in Java or Object C, are beyond what `Facebook` can achieve with the traditional web interface, through JavaScript and HTML.

Security situations. The pervasive use of WebView and mobile devices has actually changed the security landscape of the Web. For many years, we were accustomed to browsing the Web from a handful of familiar browsers, such as IE, Firefox, Chrome, Safari, etc, all of which are developed by well-recognized companies, and we trust them. Such a paradigm has been changed on smartphones and tablets: thanks to Android's `WebView` and Apple's `UIWebView`, apps can now become browsers, giving us hundreds of thousands "browsers". Most of them are not developed by well-recognized companies, and their trustworthiness is not guaranteed.

A Browser is a critical component in the Trusted Computing Base (TCB) of the Web: Web applications rely on browsers on the client side to secure their web contents, cookies, JavaScript code, and HTTP requests. The main reason why we use those selected browsers is that we trust that they can serve as a TCB, and that their developers have put a lot of time into security testing. When shifting to those unknown "browsers", the trust is gone, and so is the TCB. We do not know whether these "browsers" are trustworthy, whether they have been through rigorous security testing, or whether the developers even have adequate security expertise. Therefore, WebView has weakened the TCB of the Web infrastructure.

Another important security feature of browsers is sandbox, which contains the behaviors of web pages inside the browsers, preventing them from accessing the system resources or the pages from other origins. Unfortunately, to support better interactions between apps and web pages, WebView allows apps to punch "holes" on the sandbox, creating a lot of opportunities for attacks.

Overview of our work and contribution. Our work is the first systematic study on the security problems of WebView. The objective of this work is to conduct a comprehensive and systematic study of WebView's impact on web security, with a particular focus on identifying its fundamental causes. Through our systematic studies, we classified some existing concerns that have been raised by the community [3, 5–7] and the new attacks that are discovered by us, based on the cause of the vulnerabilities. These attacks reveal a fundamental problem caused by the weakening of the TCB and sandbox in the WebView infrastructure. Attacks are possible if the apps themselves are malicious, or if they are non-malicious but vulnerable. Android applications and web applications may become victim if they use WebView or are loaded into WebView.

Although we have not observed many attacks related to the relatively new WebView technology, it is just a matter of time before we see them on a large scale. Based on the number of applications that use WebView and the number of people who have downloaded those applications, as we will show in the case study section later, we believe that the impact is quite significant. Both Google's Android and Apple's iOS are vulnerable. In this paper, due to page limitation, we only focus on Android, but we have successfully achieved similar attacks on iOS.

Based on our studies, we discuss how we can continue benefiting from WebView, and at the same time reduce the risks. Since this paper primarily focuses on the attacks and the development of the solutions is still on going, we will leave the solution details to our future paper.

2. SHORT TUTORIAL ON WEBVIEW

In this paper, we will only focus on the Android platform. We first give a brief tutorial on Android's WebView component. On the Android platform, WebView is a subclass of `View`, and it is used to display web pages. Using WebView, Android applications can easily embed a powerful browser inside, using it not only to display web contents, but also to interact with web servers. Embedding a browser inside Android application can be easily done using the following example (JavaScript is disabled by default within WebView):

```
WebView webView = new WebView(this);
webView.getSettings().setJavaScriptEnabled(true);
```

Once the WebView is created, Android applications can use its `loadUrl` API to load a web page if given a URL string. The following code load the `Facebook` page into WebView:

```
webView.loadUrl("http://www.facebook.com");
```

What makes WebView exciting is not only because it serves simply as an embedded browser, but also because it enables Android applications to interact with web pages and web applications, making web applications and Android applications tightly integrated. There are three types of interactions that are widely used by Android applications; we will discuss them in the rest of this section.

2.1 Event monitoring

Android applications can monitor the events occurred within WebView. This is done through the hooks provided by the `WebViewClient` class. `WebViewClient` provides a list of hook functions, which are triggered when their intended events have occurred inside WebView. Once triggered, these hook functions can access the event information, and may change the consequence of the events.

To use these hooks, Android applications should first create a `WebViewClient` object, and then tell WebView to invoke the hooks in this object when the intended events have occurred inside WebView. `WebViewClient` has already implemented the default behaviors—basically doing nothing—for all the hooks. If we want to change that, we can override the hook functions with our own implementation. Let us see the code in the following:

```
WebViewclient wvclient = New  WebViewClient() {
  // override the "shouldOverrideUrlLoading" hook.
  public boolean shouldOverrideUrlLoading(WebView view,String url){
    if(!url.startsWith("http://www.facebook.com")){
       Intent i = new Intent("android,intent.action.VIEW",
                             Uri.parse(url));
       startActivity(i);
```

```
    }
  }
  // override the "onPageFinished" hook.
  public void onPageFinished(WebView view, String url) { ...}
}
          webView.setWebViewClient(wvclient);
```

In the example above, we override the `shouldOverrideUrl-Loading` hook, which is triggered by the navigation event, i.e., the user tries to navigate to another URL. The modified hook ensures that the target URL is still from `Facebook`; if not, the WebView will not load it; instead, the system's default browser will be invoked to load the URL. In the same example, we have also overridden the `onPageFinished` hook, so we can do something when a page has finished loading.

2.2 Invoke Java from Javascript

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`. Android applications can register Java objects to WebView through this API, and all the public methods in these Java objects can be invoked by the JavaScript code from inside WebView.

In the following example, two Java objects are registered: `FileUtils` and `ContactManager`. Their public methods are also shown in the example. `FileUtils` allows the JavaScript code inside WebView to access the Android's file system, and `ContactManager` allows the JavaScript code to access the user's contact list.

```
wv.addJavascriptInterface(new FileUtils(), "FUtil");
wv.addJavascriptInterface(new ContactManager(), "GC");
...
// The FileUtils class has the following methods:
public int write (String filename, String data, boolean append);
public String read (filename);
...
// The ContactManager class has the following methods:
public void searchPeople (String name, String number);
public ContactTriplet getContactData (String id);
...
```

Let us look at the `FileUtils` interface, which is binded to WebView in the name of `FUtil`. JavaScript within the WebView can use `FUtil` to invoke the methods in `FileUtils`. For example, the following JavaScript code in a web page writes its data to a local file through `FUtil`.

```
<script>
  filename = '/data/data/com.livingsocial.www/' + id +'_cache.txt';
  FUtil.write(filename, data, false);
</script>
```

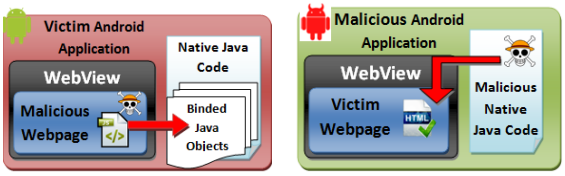
2.3 Invoke JavaScript From Java

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 called `loadUrl`. If the URL string starts with `"javascript:"`, followed by JavaScript code, the API will execute this code within the context of the web page inside WebView. For example, the following Java code adds a "Hello World" string to the page, and then sets the cookie of the page to empty.

```
String str="<div><h2>Hello World</h2></div>";
webView.loadUrl("javascript:document.appendChild(str);");
webView.loadUrl("javascript:document.cookie='';");
```

It can be seen from the above example that the JavaScript code has the same privileges as that in the web page: they can manipulate the page's DOM objects and cookies, invoke the JavaScript code within the page, send AJAX requests to the server, etc. Using `loadUrl`, Android applications can extend the functionalities of web applications, giving users a much richer browsing experience.

3. THREAT MODELS



(a) Malicious Web Pages (b) Malicious Apps

Figure 1: Threat Models

The attacks discussed in this paper are categorized based on two threat models, depicted in Figure 1. We give a high-level overview of these models in this section, leaving the attack details to later sections. It should be noted that we will not discuss the attacks that are common in the Web, such as cross-site scripting, cross-site request forgery, SQL injection, etc., because these attacks are not specific to WebView: WebView is not immune to them, nor does it make the situation worse.

Attacks from Malicious Web Pages. We study how malicious web pages can attack Android applications. In this attack model, we assume that apps are benign, and they are intended to serve a web application, such as `Facebook`. These apps can be both first-party (owned by the intended web application) and third-party (owned by an independent entity). The objective of attackers is to compromise the apps and their intended web application. To achieve this, the attackers need to trick the victim to load their web pages into the apps, and then launch attacks on the target WebView. The attack is depicted in Figure 1(a). Getting the victim to load attacker's web pages is not very difficult, and it can be done through various means, such as emails, social networks, advertisements, etc.

Attacks from Malicious Apps. We study how malicious apps can attack web applications. In this threat model, we assume that an attacker owns a malicious app, designed specifically for a web application, e.g., `Facebook`. The goal of the attacker is to directly launch attacks on the web application. The attack is depicted in Figure 1(b). Obviously, these attacks only make sense for third-party apps. To prepare for such attacks, the attacker needs to allure users to use their apps for the intended web application.

Although sounded difficult, the above goal is not difficult to achieve at all, and many apps from the Android market have already achieved that, although none of them is malicious to the best of our knowledge. For example, one of the most popular `Facebook` apps for Android is called `Friend-Caster for Facebook`, which is developed by `Handmark`, not `Facebook`; it has been downloaded for `500,000` times. The app uses WebView to browse `Facebook`.

4. ATTACKS FROM WEB PAGES

4.1 Attacks through Holes on the Sandbox

Among all WebView's APIs, `addJavascriptInterface` is probably the most interesting one. It enables web application's JavaScript code to invoke Android application's Java code (or iOS application's Objective-C code). Section 2 has already given examples on how the API is used.

Allowing apps to bind an interface to WebView fundamentally changes the security of browsers, in particular, it breaks the sandbox model adopted by all browsers. Because of the risk of running untrusted JavaScript programs inside browsers, all browsers implement an access control mechanism called *sandbox* to contain the behaviors of these programs. The sandbox basically achieves two objectives: isolate web pages from the system and isolate the web pages of one origin from those of another. The second objective mainly enforces the Same-Origin Policy (SOP).

When an application uses `addJavascriptInterface` to attach an interface to WebView, it breaks browser's sandbox isolation, essentially creating holes on the sandboxes. Through these holes, JavaScript programs are allowed to access system resources, such as files, databases, camera, contact, locations, etc. Once an interface is registered to WebView through `addJavascriptInterface`, it becomes global: all pages loaded in the WebView can call this interface, and access the same data maintained by the interface. This makes it possible for web pages from one origin to affect those from others, defeating SOP.

Opening holes on the sandbox to support new features is not uncommon. For example, in the previous Web standard, the contents in two frames with different domains are completely isolated. Introducing cross-frame communication for mashup applications to exchange data opens a hole on the sandbox. However, with the proper access control enforced on the hole, this new feature was perserved and protected. The WebView's new feature, however, was not properly designed. The objective of this paper is not against this feature, on the contrary, by pointing out where the fundamental flaw is, we can preserve Web's feature and at the same time make it secure.

Attacks on the System. We will use `DroidGap` [2] as an example to illustrate the attack. `DroidGap` is not an application by itself; it is an open-source package used by many Android applications. Its goal is to enable developers to write Android apps using mostly WebView and JavaScript code, instead of using Java code. Obviously, to achieve this goal, there should be a way to allow the JavaScript code to access system resources, such as camera, GPS, file systems, etc; otherwise, the functionalities of these apps will be quite limited.

`DroidGap` breaks the sandbox barrier between JavaScript code and the system through its Java classes, each providing interfaces to access a particular type of system resources. The instances of these Java classes are registered to WebView through the `addJavascriptInterface` API, so JavaScript code in WebView can invoke their methods to access system resources, as long as the app itself is granted the necessary permissions. The following code shows how `DroidGap` registers its interfaces to WebView.

```
private void bindBrowser(WebView wv){
   wv.addJavascriptInterface(new CameraLauncher(wv, this), "GapCam");
   wv.addJavascriptInterface(new GeoBroker(wv, this), "Geo");
   wv.addJavascriptInterface(new FileUtils(wv), "FileUtil");
   wv.addJavascriptInterface(new Storage(wv), "droidStorage");  }
```

In the code above, `DroidGap` registers several Java objects for JavaScript to access system resources, including camera, contact, GPS, file system, and database. Other than the file system and database, accesses to the other system resources need special privileges that must be assigned to an Android app when it is installed. For instance, to access the camera, the app needs to have `android.permission.CAMERA`. Once an app is given a particular system permission, all the web pages—intended or not—loaded into its WebView can use that permission to access system resources, via the interfaces provided by `DroidGap`. If the pages are malicious, that becomes attacks.

Assume there is an Android app written for `Facebook`; let us call it `MyFBApp`. This app uses `DroidGap` and is given the permission to access the contact list on the device. From the `DroidGap` code, we can see that `DroidGap` binds a Java object called `ContactManager` to WebView, allowing JavaScript code to use its multiple interfaces, such as `getContactsAndSendBack`, to access the user's contact list on the Android device.

As many Android apps designed to serve a dedicated web application, `MyFBApp` is designed to serve `Facebook` only. Therefore, if the web pages inside WebView only come from `Facebook`, the risk is not very high, given that the web site is reasonably trustworthy. The question is whether the app can guarantee that all web pages inside WebView come from `Facebook`. This is not easy to achieve. There are many ways for the app's WebView to load web pages from a third party. In a typical approach, the attacker can send a URL to their targeted user in `Facebook`. If the user clicks on the URL, the attacker's page can be loaded into WebView[1], and its JavaScript code can access the `ContactManager` interface to steal the user's personal contact information.

Another attack method is through iframes. Many web pages nowadays contain iframes. For example, web advertisements are often displayed in iframes. In Android, the interfaces binded to WebView can be accessed by all the pages inside it, including iframes. Therefore, any advertisement placed in `Facebook`'s web page can now access the user's contact list. Not many people trust advertisement networks with their personal information.

It should be noted that `DroidGap` is just an example that uses the `addJavascriptInterface` API to punch "holes" on the WebView's sandbox. As we will show in our case studies, 30% Android apps use `addJavascriptInterface`. How severe the problems of those apps are depends on the types of interfaces they provide and the permissions assigned to them.

The `LivingSocial` app is designed for the `LivingSocial.com` web site. It uses `DroidGap`, but since the app does not have the permission to access the contact list, even if a malicious page is able to invoke the `ContactManager` interface, its access to the contact list will be denied by the system. The app is indeed given the permission to access the location information though, so a malicious page can get the user's location using `DroidGap`'s `GeoBroker` interface.

Attacks on Web Applications. Using the sandbox-breaking `addJavascriptInterface` API, web applications can store their data on the device as files or databases, something that is impossible for the traditional browsers. Using `DroidGap`, the `LivingSocial` app binds a file utility object

[1]There are mechanisms to prevent this, but the app developers have to specifically build that into the app logic.

(FileUtils) to WebView, so JavaScript code in WebView can create, read/write, and delete files—only those belonging to the app—on the device. The LivingSocial app uses this utility to cache user's data on the device, so even if the device is offline, its users can still browse LivingSocial's cached information.

Unfortunately, if the LivingSocial app happens to load a malicious web page in its WebView, or include such a page in its iframe, attackers can use FileUtils to manipulate the user's cached data, including reading, deletion, addition, and modification, all of which are supported by the interfaces provided by FileUtils. As results, the integrity and privacy of user's data for the LivingSocial web application is compromised.

Like LivingSocial, many Android apps use the registered interfaces to pull web application-specific data out of WebView, so they not only cache the data, but also use Java's powerful graphic interface to display the data in a nicer style, providing a richer experience than that by the web interface. The danger of such a usage of addJavascriptInterface is that once the data are out of WebView, they are not protected by the sandbox's same-origin policy, and any page inside, regardless of where it comes from, can access and potentially modify those data through the registered interfaces, essentially defeating the purpose of the same-origin policy.

4.2 Attacks through Frame Confusion

In the Android system, interactions with several components of the system are asynchronous, and require a callback mechanism to let the initiator know when the task has completed. Therefore, when the JavaScript code inside WebView initiates such interactions through the interface binded to WebView, JavaScript code does not wait for the results; instead, when the results are ready, the Java code outside WebView will invoke a JavaScript function, passing the results to the web page.

Let us use DroidGap's ContactManager interface as an example: after the binded Java object has gathered all the necessary contact information from the mobile device, it calls processResults, which invokes the JavaScript function contacts.droidFoundContact, passing the contact information to the web page. The invocation of the JavaScript function is done through WebView's loadUrl API. The code is shown in the following:

```
public void processResults(Cursor paramCursor){
 string result = paramCursor.decode();
 string str8 = new StringBuilder().append("javascript:
               navigator.contacts.droidFoundContact(...)").
 localWebView.loadUrl(str8);
}
```

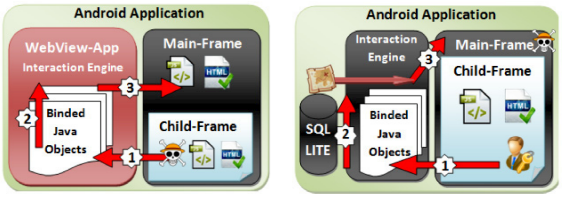
The JavaScript function contacts.droidFoundContact in the example is more like a callback function handler registered by the LivingSocial web page. The use of the asynchronous mode is quite common among Android applications. Unfortunately, if a page has frames (e.g. iframes), the frame making the invocation may not be the one receiving the callback. This interesting and unexpected property of WebView becomes a source of attacks.

Frame Confusion. In a web page with multiple frames, we refer to the main web page as the main frame, and its embedded frames as child frames. The following example demonstrates that when a child frame invokes the Java interface binded to the WebView, the code loaded by loadUrl is executed in the context of the main frame.

```
Object obj = new Object(){
  public void showDomain()
    {mWebView.loadUrl("javascript:alert(document.domain)");}
};
mWebView.addJavascriptInterface(obj, "demo");
```

The code above registers a Java object to the WebView as an interface named "demo", and within the object, a method "showDomain" is defined. Using loadUrl, this method immediately calls back to JavaScript to display the domain name of the page.

When we invoke window.demo.showDomain() from a child frame, the pop-up window actually displays the domain name of the main frame, not the child frame, indicating that the JavaScript code specified in loadUrl is actually executed in the context of the main frame. Whether this is an intended feature of WebView or an oversight is not clear. As results, the combination of the addJavascriptInterface and loadUrl APIs creates a channel between child frames and the main frame, and this channel opens a dangerous Pandora's box: if application developers are careless, the channel can become a source of vulnerability, one that does not exist in the real browsers.



(a) Attack from child frame (b) Attack from main frame

Figure 2: Threat Models

Attack from Child Frame. In this attack, we look at how a malicious web page in a child frame can attack the main frame. We use the LivingSocial app as an example. This app loads LivingSocial's web pages into its WebView (in the main frame), and we assume that one of their iframes has loaded the attacker's malicious page. This is not uncommon because that is exactly how most advertisements are embedded. The main objective of the attacker is to inject code into the main frame to compromise the integrity of LivingSocial. Web browsers enforce the Same Origin policy (SOP) by completely isolating the content of the main frame and the child frame if they come from different origins. For example, the Javascript code in the child frame (www.advertisment.com) cannot access the DOM tree or cookies of the main frame (www.facebook.com). Therefore, even if the content inside iframe is malicious, it cannot and should not be able to compromise the page in the main frame.

As we have shown earlier, LivingSocial binds CameraLauncher to its WebView. In this class, a method called failPicture is intended for the Java code to send an error message to the web page if the camera fails to operate.

```
public class CameraLauncher{
  public void failPicture(String paramString){
    String str = "javascript:navigator.camera.fail('";
    str += paramString + "');";
```

```
    this.mAppView.loadUrl(str);
  }
}
```

Unfortunately, since `failPicture()` is a public method in `CameraLauncher`, which is already binded to WebView, the method is accessible to the JavaScript code within WebView, from both child and main frames. In other words, JavaScript code in a child frame can use this interface to display an error message in the main frame, opening a channel between the child frame and the main frame. At the first look, this channel may not seem to be a problem, but those who are familiar with the SQL injection attack should have no problem inserting some malicious JavaScript code in 'paramString', like the following:

```
x'); malicious JavaScript code; //
```

As results, the malicious code embedded in `paramString` will now be executed in the main frame; it can manipulate the DOM objects of the main frame, access its cookies, and even worse, send malicious AJAX requests to the web server. This is exactly like the classical cross-site scripting attack, except that in this case, the code is injected through WebView, as illustrated in Figure 2(a).

Attack from Main Frame. In this attack, we look at how a malicious web page in the main frame can attack the pages in its child frames. We still use the `LivingSocial` as an example. We assume that the attacker has successfully tricked the `LivingSocial` app to load his/her malicious page into the main frame of its WebView. Within the malicious page, `LivingSocial`'s web page is loaded into a child frame. The attacker can make the child frame as large as the main frame, effectively hiding the main frame.

Suppose that `DroidGap` uses tokens to prevent unauthorized JavaScript code from invoking the interfaces registered to WebView: the code invoking the interfaces must provide a valid token; if not, the interfaces will simply do nothing. An example is given in the following:

```
public class Storage{
  public void QueryDatabase(SQLStat query, Token token){
    if(!this.checkToken(token))   return;
    else { /* Do the database query task and return result*/ }
  }
}
```

With the above token mechanism, even if the JavaScript code in the malicious main frame can still access the `QueryDatabase` interface, its invocation cannot lead to an actual database query. However, if the call is initiated by the `LivingSocial` web pages—which have the valid token—from the child frame, the invocation is legitimate, and will lead to a query. Unfortunately, when the query results are returned to the caller by the app, using `loadUrl`, because of the frame confusion problem, the query results are actually passed to the main frame that belongs to the attacker. This creates an information-leak channel. Figure 2(b) illustrates the attack.

5. ATTACK FROM MALICIOUS APPS

For the attacks in this section, we assume that attackers have written an intriguing Android application (e.g. games, social network apps, etc.), and have successfully lured users to visit the targeted web application servers from its WebView component.

5.1 The Problem: Trusted Computing Base

As we all know, security in any system must be built upon a solid Trusted Computing Base (TCB), and web security is no exception. Web applications rely on several TCB components to achieve security; an essential component is browser. If a user uses a browser that is not trustworthy or is compromised, his/her security with the web application can be compromised. That is why we must use trusted browsers, such as IE, Firefox, Chrome, Safari, etc.

WebView in the Android operating system changes the TCB picture for the Web, because WebView is not isolated from Android applications; on the contrary, WebView is designed to enable a closer interaction between Android applications and web pages. Using WebView, Android applications can embed a browser in them, allowing them to display web contents, as well as launch HTTP requests. To support such an interaction, WebView comes with a number of APIs, enabling Android application's Java code to invoke or be invoked by the JavaScript code in the web pages. Moreover, WebView allows Android applications to intercept and manipulate the events initiated by the web pages.

Essentially, WebView-embedding Android applications become the "customized browsers", but these browsers, usually not developed by well-recognized trusted parties but potential malicious apps, cannot serve as a TCB anymore. If a web application interacts with a malicious Android application, it is equivalent to interacting with a malicious browser: all the security mechanism it relies on from the browser is gone. In this section, we will present several concrete attacks.

However, this is differnt from the situation when attackers have compromised the whole browser by controlling the native binary code of the browser. In such a situation, attackers control everything in the browser; Malicious Android applications, however, only override the limited portion of the APIs in WebView, and the rest of WebView can still be protected by the underlying system. It is more like the usage of "iFrame", which is used to let websites embed pages from other domains; the web browser enforces the Same Origin Policy to isolate each other if they come from a different domain. Similar to the WebView situation, a malicious webpage can embed a page from Facebook into one of its iframes, the content of the Facebook page will be rendered and displayed. With the underlying access control mechanism enforced by the trusted native browser code, the Facebook page cannot be compromised by its hosting page. Similarly, if WebView is provided to applications as a blackbox (i.e no APIs), it can still be counted as a TCB component for the Web even if it is embedded into a malicious application, because isolation mechanism provided by WebView is implemented using WebKit, which is trustworthy.

5.2 Attack Methods

There are several ways to launch the attacks on WebView. We classified them in two categories, based on the WebView features that were taken advantaged of. The categories, illustrated in Figure 3, are described in the following:

- **JavaScript Injection:** Using the functionalities provided by WebView, an Android app can directly *inject* its own JavaScript code into any web page loaded within the WebView component. This code, having the same privileges as that from the web server, can

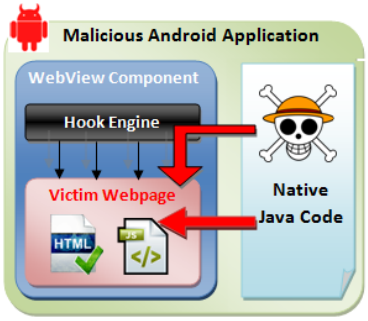


Figure 3: Attack Methods

manipulate everything in the web page, as well as steal its sensitive information.

- **Event Sniffing and Hijacking:** WebView provides a number of hooks (APIs) to Android apps, allowing them to better interact with the web page. Attackers can intercept these APIs, and launch sniffing and hijacking attacks from the outside of WebView, without the needs to inject JavaScript code.

The categories of attacking methods are presented in a decreasing order of severity: if attackers can achieve JavaScript injection, they do not need to use the second method. This indicates that some of the WebView features are more powerful than others. To fully understand the impact of WebView design on security, we study the potential attacks associated with each feature, rather than focusing only on the most powerful feature.

5.3 JavaScript Injection

Using WebView's `loadUrl()` API, Android application can inject arbitrary JavaScript code into the pages loaded by the WebView component. The `loadUrl()` API receives an argument of string type; if the string starts with "javascript:", WebView will treat the entire string as JavaScript code, and execute it in the context of the web page that is currently displayed by the WebView component. This JavaScript code has the same privileges as that included in the web page. Essentially, the injected JavaScript code can manipulate the DOM tree and cookies of the page.

WebView has an option named `javascriptenable`, with `False` being its default value; namely, by default, WebView does not execute any JavaScript code. However, this option can be easily set to `True` by the application, and after that, JavaScript code, embedded in the web page or injected by the application, can be executed.

There are many ways to inject JavaScript code into web page using `loadUrl()`. We give two examples here to illustrate the details.

JavaScript Code Injection. The following Java code constructs a string that contains a short JavaScript program; the program is injected into the web page loaded by WebView. When this program is executed in the context of the web page, it fetches additional (malicious) code from an external web server, and executes it.

```
String js = "javascript: var newscript
     = document.createElement(\"script\");";
js += "newscript.src=\"http://www.attack.com/malicious.js\";";
js += "document.body.appendChild(newscript);";
mWebView.loadUrl(js);
```

In the above example, the malicious code `malicious.js` can launch attacks on the targeted web application from within the web page. For example, if the web page is the user's `Facebook` page, the injected JavaScript code can delete the user's friends, post on his/her friends' walls, modify the user's profiles, etc. Obviously, if the application is developed by `Facebook`, none of these will happen, but some popular `Facebook` apps for Android phones are indeed developed by third parties.

Extracting Information From WebView. In addition to manipulating the contents/cookies of the web page, the malicious application can also ask its injected JavaScript code to send out sensitive information from the page. The following example shows how an Android application extracts the cookie information from a targeted web page [3].

```
class MyJS {
    public void SendSecret(String secret) {
      ... do whatever you want with the secret ...
    }
}
webview.addJavascriptInterface(new MyJS(), "JsShow");
webview.setWebViewClient(new WebViewClient() {
  public void onPageFinished(WebView view, String url){
    view.loadUrl("javascript:
        window.JsShow.SendSecret(document.cookie)");
  }
}
```

In the Java code above, the malicious application defines a class called `MyJS` with a function `SendSecret`, which receives a string as the parameter. The program then registers an instance of `MyJS` to WebView. On finishing loading the page, the application, using `loadUrl`, invokes `window.JsShow.SendSecret`, passing as the parameter whatever sensitive information the attacker wants to extract out of page. In this case, the cookie information is sent out.

5.4 Event Sniffing and Hijacking

Besides the powerful interaction mechanism between Android applications and web pages, WebView also exposes a number of hooks to Android applications, allowing them to intercept events, and potentially change the consequences of events. The `WebViewClient` class defines 14 interfaces [26], using which applications can register event handlers to WebView. When an event was triggered by users inside WebView, the corresponding handler will be invoked; two things can then be done by this handler: observing the event and changing the event, both of which can be achieved from outside of WebView without the need for JavaScript injection.

Event Sniffing: With those 14 hooks, host applications can know almost everything that a user does within WebView, as long as they register an event handler. For example, the `onLoadResource` hook is triggered whenever the page inside WebView tries to load a resource, such as image, video, flash contents, and imported css/JavaScript files. If the host application registers an event handler to this hook, it can observe what resources the page is trying to fetch, leading to information leak. Hooks for other similar web events are described in the following:

- `doUpdateVisitedHistory`: Notify the host Android application to update its visited links database. This

hook will be called every time a web page is loaded. Using this hook, Android applications can get the list of URLs that users have visited.

- `onFormResubmission`: Ask the host Android application if the browser should re-send the form. Therefore, the host application can get a copy of the data users have typed in the form.

Using WebView hooks, host applications can also observe all the keystrokes, touches, and clicks that occur within WebView. The hooks used for these purposes include the following: `setOnFocusChangeListener`, `setOnClickListener`, and `setOnTouchListener`,

Event Hijacking: Using those WebView hooks, not only can Android applications observe events, they can also hijack events by modifying their content. Let us look at the page navigation event. Whenever the page within the WebView component attempts to navigate to another URL, the page navigation event occurs. WebView provides a hook called `shouldOverrideUrlLoading`, which allows the host application to intercept the navigation event by registering an event handler to this hook. Once the event handler gets executed, it can also modify the target URL associated with the event, causing the navigation to a different URL. For example, the following code snippet in an Android application can redirect the page navigation to `www.malicious.com`.

```
webview.setWebViewClient(new WebViewClient() {
    public boolean
            shouldOverrideUrlLoading(WebView view, String url)
            { url="http://www.malicious.com";
              view.loadUrl(url); return true;
            }
};
```

The consequence of the above attack is particularly more severe when the victims are trying to navigate to an `"https"` web page, believing that the certificate verification can protect them from redirection attack. This belief is true in the DNS pharming attacks, i.e., even if attacks on DNS can cause the navigation to be redirected to a fraudulent server, the server cannot produce a valid certificate that matches with the URL. This is not true anymore in the above attack, because the URL itself is now modified (not the IP address as in the DNS attacks); the certificate verification will be based on the modified URL, not the original one.

For example, if a page within WebView tries to access `https://www.goodbank.com`, the malicious application can change the URL to `https://www.badbank.com`, basically redirecting the navigation to the latter URL. WebView's certificate verification will only check whether or not the certificate is valid using `www.badbank.com`, not `www.goodbank.com`.

Several other WebView hooks can also lead to the event hijacking attacks. Due to the page limitation and their similarity to the one discussed above, we will not enumerate them in this paper.

6. CASE STUDIES

To understand how risky the situation in Android system is, we turned our attention to the Android Market. Our goal is not to look for malicious or vulnerable apps, but instead to study how Android apps use WebView. We would like to see how ubiquitous the WebView is in Android apps, and how many apps depend on WebView's potentially dangerous features.

6.1 Sample Collection & Methodology

Apps on the Android Market are placed into categories, and we chose 10 in our studies, including Books & Reference, Business, Communication, Entertainment, Finance, News & Magazines, Shopping, Social, Transportation, and Travel & Local. We picked the top 20 most downloaded free apps in each category as the samples for our case studies.

Each Android app consists of several files, all packaged into a single APK file for distribution. The actual programs, written in Java, are included in the APK file in the form of Dalvik bytecode. We use the decompilation tool called `Dex2Jar` [4] to convert the Dalvik bytecode back to the Java source code. Due to the limitations of the tools, only 132 apps were successfully decompiled, and they serve as the basis for our analysis. We realized that `Dex2jar` has some limitations, but it was the best available tool that we could find. Since our case studies are mostly done manually, the limitations of the tool, other than reducing the number of samples, will unlikely affect our results.

6.2 Usage of WebView

We first study how many apps are actually using WebView. We scan the Java code in our 132 samples, looking for places where the WebView class is used. Surprisingly, we have found that 86 percent (113 out of 132) of apps use WebView. We plot our results in Figure 4. Percentage for each category is plotted in Figure 5.

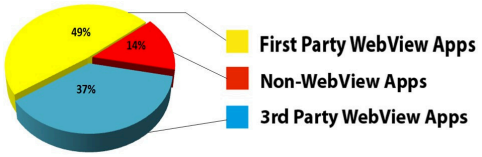


Figure 4: WebView Usage Among Apps

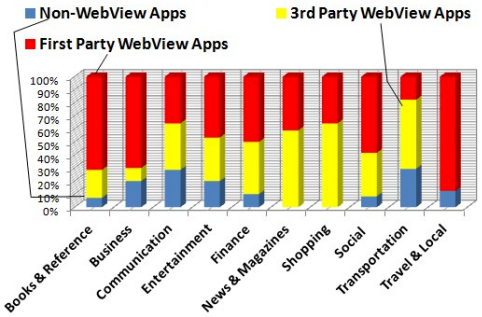


Figure 5: WebView Usage Based On Categories

For the attacks from malicious apps, it only makes sense if the apps and their targeted web applications belong to different entities, i.e., only the third-party apps have motivations to become malicious. Among the 113 apps that use WebView, 49 are third-party apps; despite the fact, these 49 apps are quite popular among users. Based on the data from the Android Market, their average rating is 4.386 out of 5, and their average downloads range from 1,148,700 to 2,813,200. Although these apps are not malicious, they are

fully capable of launching attacks on their intended web applications. When that happens, given their popularity, the damage will be substantial.

6.3 Usage of the WebView Hooks

Some of the WebView APIs are security sensitive. To understand how prevalent they have been used, especially by third-party apps, we have gathered statistics on their usage, and depict the results in Figure 6, in which we group them based on the types of attacks we discussed in Section 5.

Among the 49 third-party apps, all use `loadUrl`, 46 use `shouldOverrideUrlloading`, and 25 use `addJavascriptInterface`. We also found that the other APIs, including `doUpdateVisitedHistory`, `onFormResubmission`, and `onLoadResource` are relatively less popular. Overall, our results show that WebView's security-sensitive APIs are widely used. If these apps are malicious, the potential damages are significant.

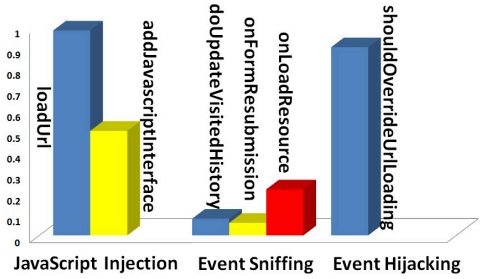


Figure 6: API Usages by Third-Party Apps

6.4 Usage of `addJavascriptInterface`

Attacks from malicious web pages are made possible by the use of the `addJavascriptInterface` API in Android apps, first-party and third-party. We would like to see how many apps actually use this API. We randomly chose 60 apps from our sample pool, decompiled them into Java code, and then searched for the usage of the API. Figure 7 depicts the results, showing that 30 percent of these apps (18 of them) do use the API.

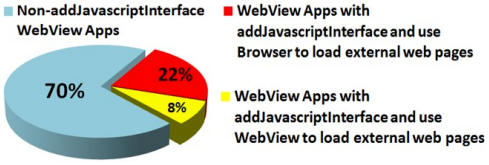


Figure 7: Source Code Investigation

Using the `addJavascriptInterface` API does not automatically make an app potentially vulnerable. To make attacks possible, attackers need to somehow get their malicious pages into the victim's WebView. This goal may not be achievable. WebView provides an hook called `shouldOverrideUrlLoading`, which is triggered every time a navigation event occurs inside WebView. Android apps can implement their own logic to process the navigation event.

Using this hook, apps can restrict what pages can be loaded into WebView, by checking whether the navigation destination URL is allowed or not; if not, they can simply change the URL, or invoke the default browser in the system to display the URL, rather than doing so in WebView. With such a mechanism, an app for `Facebook`, for example, can ensure that all the pages displayed in its WebView are from `Facebook`, essentially preventing malicious external pages from being loaded into WebView.

We have studied the 18 Android apps that use `addJavascriptInterface`, and see how they treat the navigation event. Among them, 7 use the API in the `admob` package, developed by Google for displaying advertisement. Google did a good job in restricting the WebView in `admob` to only display advertisements; if users click on one of the ads, `admob` will invoke the default Android browser to display the target page, not in its WebView. Among the rest 11, which use `addJavascriptInterface` in their own logic, 6 treat the navigation event similarly to `admob`, and the other 5 do allow their WebViews to load external web pages, making them potentially vulnerable. Our results are depicted in Figure 7.

Although using the `shouldOverrideUrlLoading` API does help apps defend against some attacks from malicious pages, it does not work if the malicious pages are inside iframes. The API is only triggered when an navigation event occurs within the main frame of the page, not the child frame. That is, even with the restriction implemented in the API, a page can still load arbitrary external pages within its child frames, making the attacks possible.

7. RELATED WORK

There are several studies focusing on Android's security architecture. The work [9] discussed potential improvement for the Android permission model, empirically analyzed the permissions in 1100 Android applications and visualized them using self-Organizing Map. However, since the attacks we proposed in the paper is not due to the flaw of security model of Android, assigning applications the least of privilege cannot prevent the attacks but only mitigate the impact of the attacks because of the limited privileges granted to the application.

Enck et al. proposes the Kirin security service for Android, which performs lightweight certification of applications to mitigate malware at installation time [12]. Enck et al. also propose "TaintDroid", an efficient, system-wide dynamic taint tracking and analysis system capable of simultaneously tracking multiple sources of sensitive data [11]. Felt et al. have built a tool called "Stowaway", which automatically detects excess privilege when installing third-party Android applications [13]. A systematic analysis of the threats in the Android Market was conducted by [27]. Those studies deal with the fact that mobile systems frequently fail to provide users with adequate control over and visibility into how third-party applications use their private data. The focus of our work is different from these studies: we focus on the security problems of WebView.

With mobile browsers playing more and more important roles in telecommunication [22], browsers themselves have become an active area of research. Microbrowsers designed for surfing the Internet on mobile device become have more and more popular [14]. Initially, research would focus on how to optimize Web content to be effectively rendered on mobile browsers [15, 21]. Recently, a lot of work has focused on analyzing the existing mobile browser models and proposing multiple new models. The paper [28] discusses two patterns

of full browsers and C/S framework browsers, and proposes a new collaborative working styles for mobile browsers. The work [24] presents a proxy-based mobile web browser with rich experiences and better visual quality.

Due to the extended use of WebView in Android applications, several Android books [19, 23] have chapters introducing how to use WebView, although none has addressed the security problems of WebView. Some discussions on WebView's security problems can be found at mainstream security-related website like ZDNet [7], and the most relevant discussions were published as blogs [3, 5, 6].

There are numerous studies that focus on enforcing fine-grained access control at the client side, including Caja [1, 10], ConScript [20], Content Security Policy [25], Escudo [16], Contego [17], work by Maffeis at al. [18], etc. Although they were not targeting the problems with WebView, some of their ideas can be extended to defend the attacks on WebView. We will pursue these ideas in our future work.

8. CONCLUSION AND FUTURE WORK

The WebView technology in the Android system enables apps to bring a much richer experience to users, but unfortunately, at the cost of security. In this paper, we have discussed a number of attacks on WebView, either by malicious apps or against non-malicious apps. We have identified two fundamental causes of the attacks: weakening of the TCB and sandbox. Although we have not observed any real attack yet, through our case studies, we have shown that the condition for launching these attacks is already matured, and the potential victims are in the millions; it is just a matter of time before we see real and large-scale attacks.

In our on-going work, we are developing solutions to secure WebView. Our goal is to defend against the attacks on WebView by building desirable security features in WebView.

9. ACKNOWLEDGMENT

We thank the anonymous reviewers and our shepherd, Patrick Traynor, for many detailed and helpful comments.

10. REFERENCES

[1] Caja. `http://code.google.com/p/google-caja/`.

[2] Droidgap. `http://www.phonegap.com`.

[3] Extracting html from a webview. `http://lexandera.com/2009/01/extracting-html-from-a-webview/`.

[4] A tool for converting android's .dex format to java's .class format. `http://code.google.com/p/dex2jar`.

[5] Injecting javascript into a webview. `http://lexandera.com/2009/01/injecting-javascript-into-a-webview/`, 2009.

[6] Intercepting page loads in webview. `http://lexandera.com/2009/02/intercepting-page-loads-in-webview/`, 2009.

[7] Researchers expose android webkit browser exploit. `http://www.zdnet.co.uk/news/security-threats/2010/11/08/researchers-expose-android-webkit/-browser-exploit-40090787/`, November 2010.

[8] U.S. smartphone market: WhoâĂŹs the most wanted? `http://blog.nielsen.com/nielsenwire/`, 2011.

[9] David Barrera, H. G üne ş Kayacik, Paul C. van Oorschot, and Anil Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 73–84, New York, NY, USA, 2010. ACM.

[10] D. Crockford. ADSafe. `http://www.adsafe.org`.

[11] W. Enck, P. Gilbert, B. G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[12] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 235–245, New York, NY, USA, 2009. ACM.

[13] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified, 2011.

[14] E. A. Hernandez. War of the mobile browsers. *IEEE Pervasive Computing*, 8:82–85, January 2009.

[15] A. Jaaksi. Developing mobile browsers in a product line. *IEEE Software*, 19:73–80, 2002.

[16] K. Jayaraman, W. Du, B. Rajagopalan, and S. J. Chapin. Escudo: A fine-grained protection model for web browsers. In *Proceedings of the 30th International Conference on Distributed Computing Systems (ICDCS)*, Genoa, Italy, June 21-25 2010.

[17] T. Luo and W. Du. Contego: Capability-based access control for web browsers. In *TRUST'11*, 2011.

[18] S. Maffeis, J. C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *IEEE Symposium on Security and Privacy*, 2010.

[19] D. McMahon. Learn android programming, 2011.

[20] L. A. Meyerovich and B. Livshits. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *IEEE Symposium on Security and Privacy*, pages 481–496, 2010.

[21] M. Palviainen and T. Laakko. Mimeframe - a framework for statically and dynamically composed adaptable mobile browsers. 2006.

[22] F. Reynolds. Web 2.0-in your hand. *IEEE Pervasive Computing*, 8:86–88, January 2009.

[23] S. Hashimi S. Komatineni, D. MacLean. Pro android 3, 2011.

[24] H. Shen, Z. Pan, H. Sun, Y. Lu, and S. Li. A proxy-based mobile web browser. In *Proceedings of the international conference on Multimedia*, MM '10, pages 763–766, New York, NY, USA, 2010. ACM.

[25] S. Stamm, B. Sterne, and G. Markham. Reining in the web with content security policy. In *WWW*, 2010.

[26] Android Development Team. Webviewclient hooks list. `http://developer.android.com/reference/android/webkit/WebViewClient.html`.

[27] T. Vennon and D. Stroop. Threat analysis of the android market, 2010.

[28] P. Ye. Research on mobile browser's model and evaluation. *Structure*, pages 712–715, 2010.