A View To A Kill: WebView Exploitation
Extended Abstract

Matthias Neugschwandtner
Secure Systems Lab
Vienna University of Technology
Email: mneug@iseclab.org

Martina Lindorfer
Secure Systems Lab
Vienna University of Technology
Email: mlindorfer@iseclab.org

Christian Platzer
Secure Systems Lab
Vienna University of Technology
Email: cplatzer@iseclab.org

Abstract—**WebView is a technique to mingle web and native applications for mobile devices. The fact that its main incentive requires making data stored on, as well as the functionality of mobile devices, directly accessible to active web content, is not without consequences to security.**

In this paper, we present a threat scenario that targets WebView apps and show its practical applicability in a case study of selected apps. We further show results of our examination of over 287,000 apps in regard to WebView-related vulnerabilities.

I. INTRODUCTION

With the rise of Web 2.0 and its technologies, the web shifted from static to dynamic content, enabling the advent of social networks and peaking in the current state of web apps that strive to rival their full-blown desktop counterparts. Parallel to this development, another sector enjoys undiminished growth: smartphones and their mobile device siblings, i.e., tablets. Inevitably accompanied by these trends is the fact that web content consumption shifts from desktop computers to mobile devices.

On mobile devices, end-users expect functionality to be delivered as a standalone app. In order to make the life for developers easier, all major mobile platforms, such as Android, iOS, Windows Phone and Blackberry introduced *WebView*. WebView is essentially a browser-library that enables developers to deliver web content, or even a whole web application as part of their smartphone client app. It is geared towards ease of use: fetching and displaying web content is a matter of a single method invocation. Using WebView, the developers do not need to re-implement and maintain their web application for every single platform. In addition, updates are distributed instantaneously and without requiring any user interaction: the developer just needs to change the content delivered by the web server.

While a pure browser-based solution would feature the same benefits, the main advantage of choosing WebView is the streamlined integration of device functionality. By making persistent storage, access to the short message service and other functionality available to the web application, the resulting apps are both flexible like web applications and powerful like ordinary applications. Typically, the developer exposes the needed APIs via a JavaScript-interface that can then be accessed from within web application JavaScript code.

The security implications of this feature are obvious: by providing a direct bridge between web content and the operating system, WebView punches a hole in the browser sandbox containment. If an attacker manages to serve malicious content to a WebView-enabled app, she will have access to all APIs that have been exposed via JavaScript.

Previous work in this area is scarce, Luo et al. [1] pick up attack vectors on WebView (as does [2]), but do not delve into the actual exploitation of apps. Bhavani [3] discusses an orthogonal problem on how a malicious app may harm a benign web page via WebView. Finally, Fahl et al. reveal orthogonal security problems in Android's SSL handling [4].

In this paper, we discuss two realistic threat scenarios that target WebView. We continue by presenting case studies on apps that we have successfully exploited. Based on the insights of the case studies, we conducted an analysis of over 287k Android apps to check for WebView-related vulnerabilities.

II. THREAT SCENARIO

A fundamental requirement for exploiting a WebView app is to gain control over the web content that is requested by the app. To access the exposed APIs, the attacker needs to inject JavaScript code that is subsequently executed by the app. Depending on time and location of the manipulation, we can distinguish between two possibilities:

Server compromise. If the attacker manages to manipulate the content stored on the server, the attack leverage is very high, since every single installation of the targeted app will be affected. The server compromise can be achieved by arbitrary means, as long as parts of the web content can be manipulated – a typical example being a stored cross-site scripting or SQL injection attack (see Figure 1(a)). A great advantage of this attack vector is that the attacker does not need to take encryption into account, as the server will take care of it.

Traffic compromise. While compromising a tightly secured server might prove difficult, manipulating the traffic on its way can be an equally capable alternative. In a typical man-in-the-middle (MITM) attack, the adversary injects the malicious code in transmitted HTML or JavaScript (see Figure 1(b)).

With mobile devices, a typical MITM attack intercepts the WiFi traffic. This can be achieved by setting up a roque WiFi access point that lures victims into connecting to them blindly. For example, the Jasager firmware of the WiFi pineapple [5] will respond to any WiFi SSID scan request and impersonate the requested network in the following.

Obviously, while the MITM attack works well with plaintext, end-to-end encryption, such as HTTPS, is an issue. Since our scenario does not include direct control over the device or the app code itself, a MITM attack will only work if the

app does not check certificate origins. In this case, the attacker can establish two encrypted channels, one to the web content server and one to the app, using a self-signed certificate.

Once the means to inject JavaScript code has been established, the actual exploit can be crafted. Its design depends on both the targeted app and platform.

On Android, APIs can be exposed as a whole: after an invocation of `WebView.addJavascriptInterface (<object>, <js_object_name>)`, the native Java object will be available through JavaScript via the provided name. The only information an attacker requires from the app in this case is the JavaScript object name. Once determined, the latter opens up vast possibilities: Via reflection the attacker can create objects and invoke their methods as long as the app has requested the corresponding Android permissions. An even more drastic example would be to use Java's `HttpClient` to download a binary executable that then runs a root exploit (e.g. rage against the cage [6]) to escalate its privileges and circumvent the permission system altogether. Naturally, such an attack would have to cope with different devices and versions to be effective.

If an app is built using the Cordova [7] framework or its predecessor, Phonegap, exploitation is even easier. Cordova is a convenience layer that sits above WebView and provides certain JavaScript interfaces, e.g. access to contacts or the camera out of the box. In addition, it always registers a JavaScript interface object called `_cordovaNative` that can be leveraged as described above.

On iOS, the attacker's possibilities are more limited, as iOS' WebView implementation does not include a "native" JavaScript bridge. Instead, most apps implement their own bridging techniques. However, a generic Cordova exploit can, for instance, always read the contact list by accessing Cordova's `navigator.contacts` object.

Generally, attacks that target frameworks such as Cordova are both app- and platform independent as long as they stick to the facilities supported by the framework and provided that the app is granted the corresponding permissions.

III. CASE STUDY

For our case study we manually analyzed and exploited four representative apps that use WebView. As a test setup we had our mobile devices connect to our own WiFi hotspot that rerouted all traffic through the mitmproxy [8].

Take Weather. This is a photo sharing app with the idea of combining weather reports on certain geographical locations with up-to-date pictures taken by the app's users. It is built based on Cordova and available for both Android and iOS. The network communication consists of JSON encoded information on the supported locations as well as the terms of use in HTML format and a JavaScript that dynamically fetches CSS style information. Since the traffic is transmitted unencrypted using plain HTTP, we can easily inject our malicious JavaScript code. On both Android and iOS we were able to access the address book, location information and the call log. On Android we could also access other Java objects via the reflection attack described above.

(a) Server compromise.

(b) Traffic compromise.
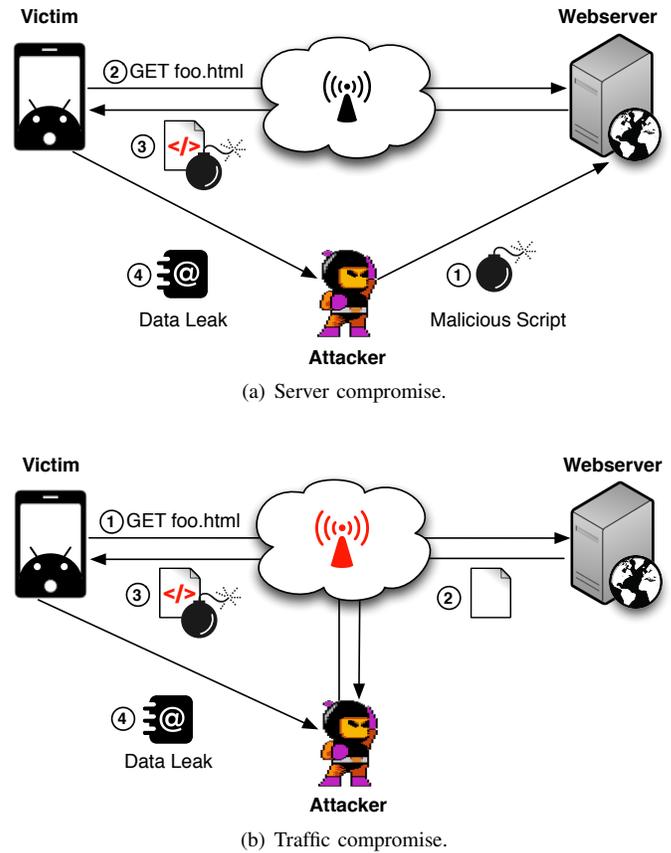
Fig. 1. Example of an attacker compromising (a) the server or (b) the traffic to steal a victim's address book.

Most Wanted. This app displays information on the most wanted criminals and terrorists of the United States. The requested permissions include access to camera and geolocation in order to be able to submit tips. WebView is used to directly display HTML content fetched from http://mobileweb.cdc. nicusa.com/most_wanted_web/. It adds a JavaScript interface to allow HTML elements to change the displayed content via a native Java object. Since the data is transmitted in plain text, it is easy to inject malicious JavaScript embedded in a `<script>` tag.

Nature Wallpaper. Who would expect harm from an app that displays nature wallpapers, has excellent ratings and features over 500,000 installs? The problem with Nature Wallpaper is that it uses a JavaScript interface to set, download and manage favorite wallpapers. Again, the traffic is unencrypted and malicious JavaScript can thus be easily injected. Since the app has the permission to access persistent storage, downloading (and executing) further malicious content would be possible.

Jiepang. This Chinese location-based social networking app offers a "check in" service similar to Foursquare and has excellent ratings as well as over 100,000 installs. In contrast to the previous applications, the traffic is partially encrypted. However, it overwrites the default WebView SSL error handler and opens the door for attackers: Its custom implementation of the `onReceivedSslError` does not perform any error handling and simply calls `handler.proceed()`, thus accepting any certificate and loading a page without notifying the

user. This circumstance and the use of a JavaScript interface exposes the app to the traffic compromise threat scenario through a (MITM) attack even in spite of the use of HTTPS. The app itself has the permissions to access persistent storage and install packages, which again would allow downloading and executing further malicious code.

IV. LARGE SCALE EVALUATION

Motivated by the results of our small case study, we proceeded to the next level: To get a grip on how widespread vulnerable WebView apps are, we examined 287,512 Android apps that had been submitted to Anubis [9] from July 2012 to March 2013.

TABLE I.   WEBVIEW USAGE

Method call	Samples	Percentage of all samples
loadUrl	166,751	58%
setJavascriptEnabled	158,042	55%
addJavascriptInterface	87,079	**30%**

WebView usage. First, we statically analyzed how many samples of our dataset perform the necessary method invocations to allow for exploitation (see Table I). Starting point is the `loadUrl` call, which fetches web content from a given URL. As a next step, `setJavascriptEnabled` has to be called with the boolean value "true" in order to enable execution of JavaScript. To finally expose a native Java object via JavaScript, `addJavascriptInterface` must be called. While well above half of the apps in our dataset fetch web content using WebView, still some remarkable 30% use a Java to JavaScript bridge functionality in their app, making it vulnerable to attacks.

TABLE II.   UNENCRYPTED HTTP APP TRAFFIC

Traffic type	Samples	Percentage of samples with a JS interface
HTML	22,803	26.18%
JavaScript	11,870	14.63%
Total	23,048	26.47%

App traffic. In theory, WebView might be used to just render web content that is stored on the device, thus making injection of JavaScript code based on our threat scenario impossible. Therefore we also analyzed the traffic transmitted during dynamic analysis in Anubis. Table II shows the results on unencrypted HTTP traffic. If either HTML or JavaScript or both are contained in the traffic, it is highly likely that an injection attack would be successful. Note that the given numbers are a lower bound, as some apps might require complex user interaction (such as a login) before they can be used and transmit network traffic.

Since end-to-end encryption makes a MITM attack impossible, we also had a look at apps that use a custom implementation of the `onReceivedSslError` handler. Developers have to overwrite this method of the WebView client to accept self-signed certificates and as we have seen in the case study of Jiepang, custom implementations can be rather "simple". Table III shows that a considerable amount of samples implements a custom WebView certificate handling. To assess their complexity, we have disassembled every custom SSL handler. The result is rather shocking: over 60% of the implementations

TABLE III.   WEBVIEW CERTIFICATE HANDLING

Certificate handling	Samples	Percentage of all samples
Custom SSL handling	10,175	3.54%
Simple SSL handler	6,208	2.16%

are "simple": without executing any conditional statement, they call `handler.proceed` right away.

Vulnerable apps. Based on the previous analysis results, we define an app as being vulnerable, if it implements a JavaScript bridge and either transmits data unencrypted or via an SSL connection that will accept self-signed certificates. According to this definition, 27,731 samples (i.e. nearly 10% of the dataset) are vulnerable. However, not every vulnerable app is equally worth to be exploited: the gain of a successful exploitation is limited by what the app is allowed to do according to its permission set. Table IV gives an overview on how many security critical permissions are granted to vulnerable apps. We have categorized the permissions into multiple groups based on which risks are associated with them.

An impressive 76% of the vulnerable samples request privacy critical permissions. Nearly 2,000 samples request the `SEND_SMS` permission that could be abused to generate revenue by sending messages to premium numbers. Finally over 60% of the samples have the necessary permission to store and run further malicious content.

TABLE IV.   PERMISSIONS OF VULNERABLE APPS

Permission (group)	Samples	Percentage of vulnerable samples
RECEIVE_SMS	1,375	4.96%
READ_SMS	1,590	5.73%
WRITE_SMS	933	3.36%
SEND_SMS	1,981	7.14%
SMS permissions	3,124	11.27%
PROCESS_OUTGOING_CALLS	355	1.28%
CALL_PRIVILEGED	134	0.48%
PHONE_CALL	0	0%
Call permissions	382	1.38%
WRITE_EXTERNAL_STORAGE	16,711	60.26%
INSTALL_PACKAGES	1,241	4.48%
Installation permissions	16,727	60.32%
READ_PHONE_STATE	18,935	68.28%
READ_CONTACTS	3,304	11.91%
ACCESS_FINE_LOCATION	11,022	39.75%
ACCESS_COARSE_LOCATION	12,923	46.60%
Privacy permissions	21,197	76.44%

Libraries. By using third party libraries that employ WebView, developers may unintentionally make their apps susceptible to our threat scenario.

As we have already mentioned, frameworks such as Cordova and its predecessor Phonegap add a JavaScript bridge with a known object name per default. If an app uses unencrypted HTTP or self-signed certificates, it is thus immediately vulnerable to a generic exploit written for the framework it employs. In our dataset, 1,435 samples use Cordova and 3,881 use Phonegap. Among those, 1,111 samples (0.39%) are vulnerable according to our definition.

But Cordova and Phonegap are not the only examples of libraries that use WebView. Table V shows to which extent third-party libraries are used by the samples in our dataset. Most of the libraries are related to ad networks while some (e.g. Flurry) collect statistics to generate revenue. We list the

top ten ad networks according to Appbrain [10] as well as the Flurry Analytics library, Greystripe and Jumptap from [11].

To assess whether they are safe according to our threat scenario, we have downloaded the current SDKs of all libraries and inspected their class files. Since a JavaScript interface is a precondition for the threat scenario on Android, we regard all libraries that do not make use of a JavaScript bridge, safe.

While with app development frameworks such as Cordova and Phonegap, the developer can still decide whether to use encryption and which resources to fetch, ad libraries function autonomously to a large extent. For example, Startapp receives the URL of the ad to click on via a JSON object. This URL is then directly used in a `loadUrl` call, which opens a JavaScript enabled WebView. The latter features a JavaScript interface named `startappwall`, whose corresponding Java object is used to report back to the library when the displayed ad is closed.

A full security audit would be necessary to evaluate whether the listed ad libraries that use a JavaScript bridge are truly safe. However, such an audit is out of the scope of this paper.

TABLE V.   LIBRARY USAGE

Library	Samples	Percentage of all samples	Safe?
Cordova	1,435	0.50%	-
Phonegap	3,381	1.35%	-
Admob	70,987	24.69%	✓
AirPush	13,462	4.68%	✓
Flurry	9,838	3.42%	✓
Millennial Media	8,663	3.01%	✓
MobClix	7,285	2.53%	?
LeadBolt	6,195	2.16%	?
InMobi	3,924	1.37%	?
Greystripe	1,787	0.62%	?
Chartboost	1,052	0.37%	✓
Jumptap	482	0.17%	?
Startapp	81	0.03%	?

V.   MITIGATION

Even if disabling JavaScript is not an option, a WebView-based app can still be hardened against attacks.

The obvious way to thwart traffic tampering is a complete end-to-end encryption. While many developers make use of HTTPS to secure their connections, they are generally reluctant to invest the money for a certificate issued by a trusted authority. Instead, they usually overwrite the default behavior of current WebView implementations and accept self-signed certificates. Consequently, these applications are prone to MITM attacks again if they do not employ countermeasures.

Such countermeasures could for example include origin checks that will drop requests that do not match a certain IP address or are not encoded using a predefined SSL certificate. Simple checks are usually implemented by overwriting the corresponding WebView handler methods [12].

On the operating system side, Android 4.2 has introduced a new annotation `@JavascriptInterface` that needs to be added to each method that is exposed via the JavaScript bridge. This effectively prevents reflection-based attacks. However, currently only 2.3% of all Android devices run version 4.2, with most devices still operating on Gingerbread [13].

To limit the harm that can be done once an app is actually exploited, the principle of least privilege should be followed, i.e. in the case of Android, only necessary permissions should be requested. Besides, Android WebView allows to separately turn off access to local storage through the JavaScript bridge.

VI.   CONCLUSION

In this short paper we have pointed out deficiencies in real-world apps that use WebView and analyzed over 287,000 samples based on our threat scenario.

In a nutshell, the benefit of a better user-experience comes at the cost of serious security implications. In case of a server compromise of just a single, vulnerable app, the consequences can be severe: seemingly harmless, simple apps like the Nature Wallpaper in our case study can exceed 500,000 installs. The resulting multiplication effect is enormous: by compromising only one server, the attacker gains access to a huge number of mobile devices.

Consequently, WebView's JavaScript support should be used with extreme caution. In order to keep app development with WebView easy, developing a static code checking tool for WebView related vulnerabilities could be rewarding future work.

REFERENCES

[1] 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 (ACSAC)*, 2011.

[2] "Abusing WebView JavaScript Bridges," http://50.56.33.56/blog/?p=314, December 2012.

[3] A. Bhavani, "Cross-site Scripting Attacks on Android WebView," *International Journal of Computer Science and Network*, vol. 2, no. 2, 2013.

[4] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith, "Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS)*, 2012.

[5] "WiFi Pineapple," http://hakshop.myshopify.com/products/wifi-pineapple, last accessed July 2013.

[6] "Rage against the cage," http://thesnkchrmr.wordpress.com/2011/03/24/rageagainstthecage/, March 2011.

[7] "Apache Cordova," http://cordova.apache.org.

[8] "Man-in-the-middle proxy," http://mitmproxy.org.

[9] "Anubis," http://anubis.iseclab.org.

[10] "Android ad networks," http://www.appbrain.com/stats/libraries/ad.

[11] S. Shekhar, M. Dietz, and D. S. Wallach, "AdSplit: Separating Smartphone Advertising from Applications," in *Proceedings of the 21st USENIX Security Symposium*, 2012.

[12] "Adventures with Android WebViews," http://labs.mwrinfosecurity.com/blog/2012/04/23/adventures-with-android-webviews/, April 2012.

[13] "Android Platform Distribution," http://developer.android.com/about/dashboards/index.html.