Bifocals: Analyzing WebView Vulnerabilities in Android Applications

Erika Chin and David Wagner

University of California, Berkeley
{emc, daw}@cs.berkeley.edu

Abstract. WebViews allow Android developers to embed a webpage within an application, seamlessly integrating native application code with HTML and JavaScript web content. While this rich interaction simplifies developer support for multiple platforms, it exposes applications to attack. In this paper, we explore two WebView vulnerabilities: *excess authorization*, where malicious JavaScript can invoke Android application code, and *file-based cross-zone scripting*, which exposes a device's file system to an attacker.

We build a tool, Bifocals, to detect these vulnerabilities and characterize the prevalence of vulnerable code. We found 67 applications with WebView-related vulnerabilities (11% of applications containing WebViews). Based on our findings, we suggest a modification to WebView security policies that would protect over 60% of the vulnerable applications with little burden on developers.

Keywords: Security, smartphones, mobile applications, static analysis.

1 Introduction

Mobile devices and platforms are a rapidly expanding, divergent marketplace. Application developers are forced to contend with a multitude of Android mobile phones and tablets; customized OS branches (e.g., Kindle Fire, Nook Tablet); and a score of competing platforms including iOS and Windows Phone. Android developers are responding to the challenge of supporting multiple platforms through the use of WebViews, which allow HTML content to be displayed within an application. At a high level, WebViews provide the same functionality as a web browser, but allow full customizability with respect to how and what content is displayed (e.g., navigation UIs, full screen, etc). These in-application browsers allow developers to write code in platform-neutral HTML and JavaScript that can be displayed by any device and version. Furthermore, application updates become simple. Developers merely update the HTML content downloaded by an application.

While convenient, these customized browsers can also pose a threat to application security, as allowing web content to interact with the application increases the application's attack surface. We show in this paper that these problems are real.

One feature of Android is that it provides a way for JavaScript in a WebView to invoke Android application code, if this is enabled by the application. In particular, the application developer can register an interface (an API to the mobile application) that can be called by the JavaScript. This allows the web page to access functionality and data exposed by the application. This may seem safe, as typically developers use WebViews to display trusted websites. However, it introduces a new risk [29]. If the user

navigates the WebView to an untrusted malicious website, the malicious page may receive access to potentially sensitive application data. Similarly, if the application loads a page over HTTP and if the user is using an insecure WiFi network, a man-in-the-middle could inject malicious content into the page and mount a similar attack. Allowing JavaScript to invoke application code breaks traditional browser security models.

In this work, we detail various WebView-based attacks and present our vulnerability identification tool, Bifocals. We ran the tool on a data set of 864 applications. Among the 608 applications that contain WebViews, we find that over 20% of applications have the potential to give websites access to code. Of these applications, we find 54% allow a user to navigate to malicious JavaScript that could access application code.

Based on our findings, we recommend modifications to Android to address these risks. Our experiments suggest that these modifications would protect more than 60% of the vulnerable applications.

We make the following contributions:

- We build a tool to identify vulnerable WebViews.
- We measure the prevalence and impact of vulnerable WebViews.
- We suggest and evaluate solutions to mitigate these vulnerabilities.

2 Application and Web Interaction

To understand vulnerabilities in WebViews, we must first understand the features provided by WebViews. The WebView class allows developers to display data from web pages and files within the confines of the application, seamlessly integrating web and application content. Through the WebView, not only can developers set the content to be displayed, but they can also specify the layout and behavior of the WebView (e.g., display the address bar, track the browsing history, allow searches, etc.). Essentially, the WebView class allows a developer to create their own custom, embedded web browser.

Alternatively, web content can be displayed by sending a request to a browser application to load the content. We will focus on the WebView approach to displaying web content as customizations in a WebView can lead to security problems, while browsers are separate applications outside of an application's security boundary.[1]

We discuss how WebViews are created and how they can be customized in detail.

2.1 WebView API

The WebView API allows developers to display content in various formats. WebViews can load (1) web content using the HTTP or HTTPS protocols, (2) files from the file system via "file://," and (3) HTML via "data://." By default, a basic WebView does not execute JavaScript nor can the web content interact with the application in any way. If the user clicks on a link within the WebView, the application is exited and the URI is loaded by the device's default web browser.

2.2 WebView Customizations

We discuss relevant WebView customizations that can be made by the developer. We list the APIs in Table 1.

[1] We use the term "web browser" to specifically reference a device's default web browsing application and "WebView" to refer to developer customized views.

API call
`setWebViewClient(WebViewClient client)`
`addJavascriptInterface(Object object, String name)`
`getSettings().setJavaScriptEnabled(...)`

Table 1: *Select list of API calls used to customize WebView behavior*

WebSettings (Javascript and file access). Each WebView contains its own WebSetting. The Android WebSettings class manages the settings of a WebView:

- Javascript execution in a webpage can be enabled by calling `setJavaScriptEnabled()` on the WebSetting. By default, JavaScript execution is off.
- Access to the local file system (e.g. loading a file in a WebView) is enabled by calling `setAllowFileAccess()`. By default, WebViews have file system access.[2]
- Access to files by JavaScript running in the context of a file scheme URI is enabled by calling `setAllowFileAccessFromFileURLs()`. By default, WebViews grant this access for API versions prior to Jelly Bean.
- Access to content from any origin by JavaScript running in the context of a file scheme URI is enabled by calling `setAllowUniversalAccessFromFileURLs()`. By default, WebViews grant this access for API versions prior to Jelly Bean.

WebViewClient (Navigation ability). A WebView may or may not have an associated WebViewClient. The Android WebViewClient class is an event handler that allows developers to specify how content is rendered. By subclassing this client, the developer can specify what actions should be taken when the page finishes loading, a resource is loaded, an error is received, etc. Most notably, it allows the developer to specify the navigation behavior of the WebView (i.e., what action should be taken when the user clicks on a link in the WebView.) By overriding the default `shouldOverrideUrlLoading()` method, the developer can take different actions based on the contents of the URI. For example, a developer may specify that the URI be loaded in the WebView if it is on a specific domain, otherwise it launches the URI via web browser.

The default behavior of the WebView when the user clicks on a link in the WebView depends on the WebViewClient. We show this in Table 2. A WebView without a WebViewClient launches the web browser. If the WebView has a WebViewClient, the behavior depends on the `shouldOverrideUrlLoading()` method. If the method is not overridden or it returns `false`, then URIs are launched in the WebView. Otherwise, the behavior depends on the implementation of the method.

Has WVC?	shouldOverride()?	Loaded in:
No	N/A	Browser
Yes	Default	WebView
	Returns false	WebView
	Returns true	Depends on impl.

Table 2: *How navigation events are handled, based on properties of the WebViewClient (WVC)*

[2] Regardless, access to an application's assets and resources (located at `file:///android_asset` and `file:///android_res`) is always granted within each application.

Interfaces (Code access). Developers can also give web content access to the application's internal Java code. By calling `addJavascriptInterface(Object object, String name)`, the developer provides a handle to an application's interface to be used by JavaScript in loaded pages. For example:

```
WebView wv = new WebView();
wv.getSettings().setJavaScriptEnabled(true);
wv.addJavascriptInterface(new MyClass(), "mycls");
wv.loadURL("http://www.foo.com");
```

The above code creates a WebView where its web contents can invoke methods in `MyClass`. Any webpage in the WebView can invoke the methods with this JavaScript:

```
<script>
    mycls.someMethod1();
    mycls.someMethod2();
</script>
```

WebViews provide a way to meld applications with web content. Developers can allow JavaScript to invoke registered application methods, potentially enabling application state to be altered on the fly; and they control how a user may navigate pages. These can be powerful mechanisms towards providing a rich, interactive user experience. However, they can also introduce security vulnerabilities.

3 Attacks

The use of WebViews exposes applications to a larger attack surface. We discuss two types of vulnerabilities we identified: excess authorization and file-based cross-zone scripting, and the relevant threat model for attackers to exploit these vulnerabilities.

3.1 Threat Model

We assume developers are not malicious, though they may have varying levels of expertise in developing on the Android platform. While the application itself is trusted, web content and the open network it passes over should not be. We will discuss this in greater detail as we explain each vulnerability.

3.2 Excess Authorization

When a developer enables JavaScript execution and registers interfaces to a WebView, JavaScript content in the WebView can invoke the registered interfaces. If malicious third-party JavaScript gets loaded in the page, then it too can invoke the application's registered Java code. As authorization is actually granted to more web content than intended, we call this an excess authorization vulnerability. This general attack was introduced by Luo [29]. We develop variations and design and conduct a large-scale measurement study to understand the prevalence of this vulnerability.

Repercussions Access to the application's Java code can lead to a variety of security implications depending on the functionality of the Java code. Information injection and leakage can occur if the invoked methods receives and returns information, respectively. Malformed input parameters may be able to crash the application, corrupt data, or otherwise launch a denial of service. Privilege escalation can occur if the methods require privileges that are owned by the applications [24, 22]. Malicious JavaScript, in combination with other application vulnerabilities such as inter-application messaging vulnerabilities [17], can lead to attacks on other applications installed on the device. These are just a few of the ways an attacker can wreak havoc on an application.

Attackers We consider two threat models:

Malicious third-party content. There are many ways malicious JavaScript can appear in a WebView. Usually, the first-party content on the first page loaded is trusted. However, this page could also contain ads. Malicious ads containing JavaScript have appeared on popular advertising networks such as Google, Yahoo, and The New York Times [5, 6, 13]. Another way third-party content can be embedded in the page is through the use of frames. Finally, the user may navigate to third parties via links (if allowed by the WebView's settings). If any of this third-party content is malicious, it could invoke the application interfaces in ways the developer might not have anticipated.

Network attacker. Another variation on this vulnerability is if the device is on an insecure network. If any page or resource is loaded over an unencrypted connection (i.e., over HTTP), then a man-in-the-middle attacker could inject any page of his choosing as a response to the request and thereby inject malicious JavaScript into the WebView.

Other threats not considered in this paper. Even supposedly "trusted" websites can present a threat. First, trusted parties may purposely include what they think to be benign, third-party JavaScript. Nikiforakis et al. have shown that over 88% of websites include at least one remote JavaScript library [30]. Malicious JavaScript could be include and could invoke the Android application's interface.

Additionally, "trusted" websites may also contain a cross-site scripting (XSS) vulnerability that allows an attacker to load malicious JavaScript in the page [18, 27, 20]. Over 75% of web applications are estimated to be vulnerable to cross-site scripting [33]. If a page loaded in the WebView is vulnerable to XSS, an attacker may be able to exploit the XSS vulnerability to introduce malicious JavaScript into the page and then attack the mobile application.

For the purposes of this study, we focus on malicious third-party content and network attackers. Vulnerabilities in trusted websites can be inferred by assuming that $75 - 88\%$ of websites may also pose a threat due to remote script inclusion or XSS.

3.3 File-based Cross-zone Scripting

The Android WebView renderer treats everything loaded via a "file://" URL as being in the same origin. This allows any content loaded via a "file://" URL to read any file on the filesystem that the application can, including application internal storage (which is not accessible to any other application) and, if the application has permission, any file stored on the SD card. If the application loads static content via a "file://" URL, and this content includes third-party, untrusted JavaScript (or includes JavaScript over an unencrypted HTTP connection), this JavaScript gains the ability to read all the files in the filesystem that the application can.[3]

If the JavaScript is requested over an insecure connection, a man-in-the-middle attacker can inject malicious JavaScript. If the JavaScript is requested over HTTPS, but from an external, potentially untrusted source, the JavaScript itself could be malicious. Once malicious JavaScript is loaded, it can read files, create a network connection, and send the contents back to the attacker.

[3] Caveat: In the latest release of Android, the Android OS was modified to require developers to explicitly enable access to "file://" URLs, reducing the opportunity for attack. For applications prior to Jelly Bean and for applications that do not set the minimum OS version to Jelly Bean, access to files is still granted by default.

The exposed surface for this attack is admittedly smaller than the excess authorization attack. Only loaded files provide access to the vulnerability, and once the user navigates away from the "file://" scheme, the attack can no longer be launched. Similarly, the attack cannot be launched through a non-file frame. As we find in our measurement study, file-based cross-zone scripting vulnerabilities are fortunately fairly rare.

4 Bifocals

We present a tool, Bifocals, which closely examines two aspects of WebView interaction, the application and the web content, to automatically identify WebView vulnerabilities in Android applications. In Section 4.1, we describe how we analyze Android applications to identify at-risk WebViews. In Section 4.2, we describe how we crawl and analyze the web pages loaded into WebViews, to determine whether an attacker may be able to inject malicious Javascript into the WebView. In Section 4.3, we describe how we put these parts together to determine the potential impact of an attack.

4.1 Application Analysis

The first step of the tool is to detect potential WebView vulnerabilities.

Policy. If a WebView enables JavaScript, registers a JavaScript interface, and loads a URI, then it may be vulnerable to an excess authorization attack (depending on the content loaded). WebViewClient settings determine whether a user can navigate away from the page while staying within the confines of the WebView. This increases the potential for attack because every page a user navigates could also contain malicious JavaScript, as opposed to just the initial landing page.

Implementation Details. Applications for the Android platform are comprised of Dalvik executable (DEX) files that run on Android's Dalvik Virtual Machine. We first disassemble application DEX files and extract XML content and file resources packaged with the application using the publicly available Dedexer [31] and Baksmali tools [12].

Bifocals statically analyzes the disassembled output. Static analysis is a common approach for bug finding [16, 28, 37]. Bifocals specifically performs flow-sensitive, interprocedural static analysis. For optimization purposes, we limit the method invocation tracking to a nesting depth of three. Experimentally, we have not seen any cases where WebView information is propagated more than three levels deep. Bifocals tracks the state of WebViews (and WebView subclasses), WebViewClients (and WebViewClient subclasses), strings, numbers, and any relevant fields, parameters, and return values.

For each method that uses WebViews, Bifocals determines:

1. Whether JavaScript execution has been enabled for the WebView
2. If it allows JavaScript, what interfaces are made accessible to the JavaScript
3. The URI that is being loaded
4. Whether a user can navigate to other webpages within the WebView (by evaluating the implementation of any methods that override `WebViewClient.shouldOverrideUrlLoading()`)

In most cases, these properties are determined by tracking information to the WebView (string value, numbers, classes, etc.). Determining the fourth property requires a little more explanation. In addition to implicitly setting a navigation policy via the presence of the WebViewClient or using the default behavior of the `WebViewClient.`

`shouldOverrideUrlLoading()` method, developers may also apply a policy for navigation behavior through code in the `WebViewClient.shouldOverrideUrlLoading()` implementation. We apply a heuristic to infer navigability. If the implementation of this method returns `false`, then users can navigate within the WebView. If the code for this method (or any methods called within the code for this method) contains a load URI call, then users can navigate within the WebView, *unless* it also contains a message invocation to launch the web browser. In that case, the developer has set a hybrid policy (e.g., loading the page in the WebView if the domain is `mysite.com` and launching the browser otherwise), and we conservatively consider that any new URIs will launch the browser (limiting the navigability, and thus, the attack opportunity).

These vulnerable WebViews and the URIs loaded into them are passed to the web analysis portion of the tool.

4.2 Web Analysis

The second stage analyzes the URIs (websites, files, and data) that are being accessed to determine whether they might embed or navigate to third-party content.

Policy. For each URI, Bifocals examines the page for potentially malicious third-party content. We focus specifically on attack scenarios where malicious JavaScript may be included in the WebView via website content, insecure networks, and user navigation and not via the exploitation of XSS vulnerabilities. Although third-party content can encompass many forms of content (e.g., images, scripts, frames, etc.), we limit the definition of potentially malicious third-party content to content that can lead to the execution of untrusted script. We classify ads and frames that load third-party sites as potentially malicious. Ads can be supplied by anyone and can contain JavaScript. Similarly, frames that load external content are considered untrusted. We ignore third-party images and other content that does not contain or execute script. We also ignore non-ad-related JavaScript (e.g., non-ad `<script src=...>`) unless it is embedded in a third-party page. Many webpages include popular, trusted third-party JavaScript, such as Google Analytics, Facebook's "Like" button, etc., and we assume these are intentional and we do not treat them as potentially malicious.

If a WebView is navigable, we apply the same evaluation to all pages transitively linked from the landing page (to a depth of three). Additionally, if the user can navigate to a third-party page (via links) in the WebView, we classify it as potentially malicious.

We assume that the primary website being visited and sites within the same domain are trustworthy, as well as anything belonging to the same second-level domain (the domain directly below the top-level domain in the DNS). For example, suppose a WebView loads `http://mysubdomain.mysite.com`. The domain `mysubdomain.mysite.com`, its second-level domain (SLD) `mysite.com`, and other subdomains of it (e.g., `myothersubdomain.mysite.com`) are most likely under the same jurisdiction and therefore we treat them all as trustworthy. This trust is similar to the implicit trust of cookie setting between a subdomain and its parent domain [4]. In the case of domains with country codes, we take the third-level domain. (E.g., `http://blogs.telegraph.co.uk`'s trusted domain would be `telegraph.co.uk`.)

Implementation Details. To perform this analysis, we build on a basic web crawler built as a Firefox extension [25]. Given a URI, this crawler invokes Firefox, loads

the page, and returns redirect information and the HTML source (including the frame source). We modified the extension to also log links, frames, and links within frames.

To identify ad content, we incorporated and modified the Adblock Plus extension [1]. Adblock Plus is a browser extension that parses pages and identifies and removes ads. For every network request required to load a page, it invokes a JavaScript function `shouldLoad()` that returns whether that content is an ad and should be loaded. We modified Adblock Plus in two ways. First, we modified the `shouldLoad()` function to log the content type (e.g., script, image, document, subdocument, etc. [9]), request origin, and target location. Second, we always allow the content to be loaded but log when an ad is identified.

To simulate a mobile browser, we modify the Firefox preference file (prefs.js) to set the user-agent string to the user-agent of an Android browser. This way, the web behavior returned by the request is the mobile behavior, not the desktop browser behavior.

Finally, we modify URIs before loading. For URIs that load data, we prepend the HTML with `data:text/html,` so that the browser loads the data string as a data URI. For URIs that load data with a relative base URI, we prepend the HTML with `data:text/html,<base href='' + theBase + ''>` to ensure that the browser renders the data and resolves all relative references.

The crawler then crawls the URIs that could be loaded into the WebView. If a vulnerability is identified or the WebView that the URI is from is not navigable, the crawling for that URI ends. Otherwise, the crawler repeats the page analysis for all links in the page and frames with the same SLD as the original URI or its redirects. We limit the crawling link depth to three for feasibility reasons.

Results from the crawler and the application analysis are then combined to identify WebViews that are fully vulnerable to the excess authorization attack.

We identify file-based cross-zone scripting attacks by checking if any of the loaded file URIs (regardless of whether interfaces are registered) contain third-party JavaScript.

4.3 Impact Analysis

There are many ways to examine the impact of a vulnerability. As discussed in Section 3.2, an attack on a WebView could result in information leakage, information injection, DoS, etc. One way to measure impact is to examine how many privileged resources an attacker would gain access to. We do this by analyzing the code invoked by the interface and determining the permissions required to execute that code.

We built a tool to determine what Android APIs a registered interface transitively grants access to (through invocation) and the permissions they correspond to. Given an interface, we analyze all methods that can be accessed in that interface (namely, all public methods and any superclasses' public methods). We assume that the attacker can determine public methods via reflection or direct analysis of the target application.

For each of the directly accessible methods, we recursively analyze the methods invoked by the method and the Android API calls made in the method. If an interface method returns an object of a different class, we analyze that object's public methods as well. We apply an Android API-to-permission map [21] to determine the set of permissions used by the reachable code. To determine the permissions used by non-API calls, Android message passing, Android databases, and code invoked via Java reflection, we modify Felt et al.'s Stowaway tool [21] to identify and output the methods in which these permissions are used. If those methods are reachable, then we add the cor-

responding permissions to the permission set. We include both normal and dangerous permissions in the set of permissions used.

4.4 Limitations and discussion

Platforms. There are alternatives to using Firefox extensions to perform a web crawl. We could have used a command-line tool (e.g., `wget`), however this has limitations on the information received from the page. We chose a full-featured web browser which allowed us to leverage the existing Adblock extension, parse the loaded DOM in real-time, and fully render the content.

We chose to run this on a desktop computer with modifications to the browser preferences to spoof a mobile browser, as Firefox is more robust and efficient in crawling pages at scale. Given the massive amounts of meta-data produced from the crawl (from a large data set), performing the crawl on a mobile device would present challenges of dealing with a less robust, memory- and space-limited operating system. While it is possible for websites to rely on fields other than user-agent to determine whether it is running on a mobile device (and change content accordingly), user-agent is by far the most commonly used field. In fact, we investigated the possibility of alternate indicators (e.g., JavaScript's `Navigator.platform` or `Navigator.appName`), but we observed only the user-agent being used in the websites we crawled. Even if websites were modified based on different Navigator fields, it is more likely to change the layout, not the nature of the content (frames, ads, etc.), and therefore it would not impact our results.

Ad Networks. Although we identify ads as potentially malicious, some ad networks may prohibit JavaScript from advertisers. We did not further classify ad networks based on whether a third-party advertiser could include JavaScript.

Crawling. One of the limitations of our crawling approach is the possibility of false negatives. Web content is dynamic. An ad or other third-party JavaScript may not always appear on a given page. To address this, we crawled each page three times.

Another potential source of false negatives is the inability to crawl all content. We limited the crawl depth to three links, but untrusted JavaScript may be on a page that our tool did not crawl. Websites might prevent our crawler from seeing the content behind a pay-wall or login-wall. In this case, our crawler will only analyze the login page. To address this, we would have to manually create accounts, log in, and crawl the page.

Due to these limitations, our tool reports a lower bound on vulnerable applications. On the other hand, mobile applications change less frequently than web content, and we can use the number of potential WebView vulnerabilities from the application analysis to estimate an upper bound on the number of actual vulnerable WebViews.

Static analysis. A limitation to our static analysis approach is the risk of not deriving the full URI. If a URI is comprised of strings that are obtained from dynamic messages (Intents), from an API call that we do not handle, or from system state (e.g., getting the device ID, getting accelerometer data), then static analysis may fail to infer the full URI loaded into the WebView. Crawling an invalid URI could result in a redirect to a different page. In most cases, we believe that the redirected page would also be representative of the content that the page would have displayed (in terms of using ads and linking to third-parties). We additionally supplement missing data by substituting logical default values for substrings that cannot be derived. For example, if float value that we do not track is included in the URI, then a "1.0" is inserted in its place. Our

tool also does not attempt to handle implicit control flow or resolve Java reflection of the WebView API, and this could lead to false negatives. Our tool, however, does resolve Java reflection for the impact analysis which is more likely to contain reflection. (Developers are unlikely to reflectively call the WebView API as the API is already publicly accessible.)

We considered a dynamic analysis approach to Bifocals as an alternative to our static approach. A dynamic analysis tool would be able to accurately determine dynamically set variables and state. It would also be able to confirm a vulnerability by exploiting it at run-time. However, it would be challenging to explore the full application state space to traverse all WebViews and to generate valid input for malicious JavaScript. Additionally, some Android UIs cannot be explored without user input (e.g., applications with logins). We chose a static approach because it achieves better code coverage, increasing the possibility of discovering vulnerabilities that may not have been exposed at runtime. We leave the possibility of a combined static and dynamic approach to leverage the benefits of both techniques for future work.

5 Evaluation

We ran Bifocals on 864 popular Android 2.2 applications to identify the prevalence of WebView vulnerabilities. The dataset consists of the 100 most popular paid applications, 764 most popular free applications, and 100 recently added free applications from the Android Market (as of Oct. 2010). After removing duplicate applications, applications that only consisted of keys to unlock paid features for free applications, and applications used for tool development and testing, we were left with a set of 864 applications for analysis.[4]

5.1 Characterizing the use of WebViews

Developer use of WebViews. We first analyzed these applications to better understand their use of WebViews. We found that 608 of the 864 applications (70.4%) contained at least one WebView in the application. Of these 608 applications, 433 (71.2% of applications with WebViews, 50.1% of all applications) contained at least one WebView in the core functionality of the application. Also, 351 applications (57.7% of applications with WebViews, 40.6% of all applications) contained at least one WebView displayed by an ad library in the application.[5] This suggests that use of web content in Android applications is common.

The web content displayed in a WebView can be hosted remotely or locally. We analyzed all WebViews in these applications to identify what URI is initially loaded into the WebView. In Table 3, we summarize the schemes used by these applications. Overall, many applications load content over HTTP or via the data scheme. Use of SSL is much less common.

[4] We wanted to analyze both free and paid applications in order to avoid biases that might be present in free applications. Therefore, we reused an existing dataset rather than buying the applications a second time. It would be interesting to see if the results differ if we were to repeat the same experiments on current applications.

[5] In the rest of the section, we may shorten the phrases "WebView in the core functionality of the application" to "core WebView" or "core application" and "WebView in an ad library in the application" to "ad WebView" or "ad application."

Content loaded via:	# of apps	%
HTTP or HTTPS	345	56.7%
http://	335	55.1%
https://	15	2.5%
Local static content (file/data)	374	61.5%
file://	103	16.9%
data: (e.g., <html>...)	323	53.1%

Table 3: The types of URIs loaded into WebViews

	Total	Core	Ad
Apps with WebViews	608	433	352
Apps with auth'ed WVs	120	85	38
%	19.7%	19.6%	10.8%

Table 4: Breakdown of applications that grant JavaScript code access by whether the WebView is in the core application or ad library

Exposure of interfaces. We further examined how many applications allow JavaScript to invoke application code (by registering interfaces). We call these *authorized WebViews*. As indicated in Table 4, of the 608 applications with WebViews, we find that one-fifth of these applications have at least one authorized WebView. Furthermore, one-fifth of applications have authorized, core WebViews, while 10.8% of applications have authorized, ad WebViews.[6] This suggests that many developers do use WebView APIs to grant web content access to application content. The 38 applications with authorized ad WebViews can be attributed to three distinct ad providers: Millennium Media [8], AdMarvel [2], and Medialets [7].

In Table 5, we further break these authorized WebViews down by the scheme of the URI initially loaded in the WebView. Unsurprisingly, many of these WebViews load content over the HTTP protocol, and very few use SSL. The distribution of schemes for these types of WebViews closely mirrors that for all WebViews, except that fewer of the applications loading content via data schemes expose an interface (10% vs. 16%; $p = 0.025$, Fisher's exact test).

Among the 85 applications that expose interfaces to core WebViews, 34 applications (40%) have WebViews where the user can navigate within the WebView, while 51 applications (60%) have WebViews that limit navigation (by launching subsequent URLs in a browser application). This is promising, as it shows that a majority of the applications have reduced their potential attack surface. However, restricting navigation does not fully eliminate the risk if the first page includes third-party frames or JavaScript.

5.2 Automated Analysis

In summary, Bifocals found 67 applications (11.0% of applications with WebViews, and 55.8% of applications with authorized WebViews) that are vulnerable to at least one of the attacks presented. The high rate of vulnerabilities suggests that the Android WebView interface is error-prone and exposing APIs to web content is particularly risky.

[6] The sum of the applications with core and ad WebViews exceed the 120 applications as some applications have both core WebViews and ad WebViews.

Authorized WVs by URI scheme	# of Apps
http://	57 (47.5%)
https://	2 (1.7%)
file://	19 (15.8%)
data:	32 (26.7%)

Table 5: Breakdown of authorized applications by the URI scheme used

Vulnerability	Core	Ad	Total
Network attack	32	33	65
Web attack	18	33	51
Total	33	33	66

Table 6: The number of vulnerable apps found by Bifocals.

Excess Authentication Vulnerabilities We summarize the number of vulnerable applications in Table 6. We evaluate ad and core WebViews separately, as vulnerabilities in ad libraries can only be fixed by the ad provider, while vulnerabilities in the core application can be fixed by the application developer. Also, patching one ad library could secure multiple applications while patching vulnerabilities found in core WebViews must be done individually by each affected developer.

Network attacker. We found 65 applications (54.1% of applications that register interfaces) that are vulnerable to an excess authorization attack if used while connected to an insecure network.

The impact of these vulnerabilities varies. For 18 (56.2%) of the 32 applications with this type of vulnerability in a core WebView, a network attacker gains access to API calls that use one or more Android permissions available to the application. Thus, the attacker may be able to take actions that would not be available to arbitrary web content. None of the ad libraries' WebViews give access to API calls that require permissions, so those vulnerabilities may have lower impact. It is important to note that access to permissions is only one metric to measure impact. Several other attacks may be possible even on applications whose API does not use any special permissions.

Web attacker. Bifocals found 51 applications (42.5% of applications that register interfaces) that are vulnerable to attack through malicious websites.

Many of these vulnerabilities grant a malicious website abilities that we would not expect web content to receive. 13 (72%) of the 18 applications containing a core WebView that is vulnerable to a web attack give the web attacker the ability to invoke an API that uses one or more of the application's Android permissions. In contrast, none of the ad-based vulnerabilities allow attackers to invoke code that uses permissions.

File-based Cross-zone Scripting Vulnerabilities Our tool identified two applications that load files with remote JavaScript. One of these is vulnerable to a network attack. The other makes requests over SSL from a trusted site, making it resistant to attack.

Upon further inspection, we find that many files loaded into a WebView are simple HTML pages with no need for JavaScript. For example, files may contain EULAs, Terms of Service, and FAQ pages.

5.3 Manual Analysis

We randomly selected 10 applications (of the 18 applications with a web-based excess authorization vulnerability in a core WebView) and manually analyzed these applications to determine the false positive rate of Bifocals. For each selected application, we examined the code, the loaded websites, and application as installed on an Android phone. For each reported vulnerability, we confirmed that Bifocals correctly inferred the APIs registered, URIs loaded, and navigation capability of the WebView. For each loaded URI, we confirmed the crawler result: that an ad, external frame, or site was found within the navigation constraints of the WebView. We did not build an exploit. We manually analyzed 19 vulnerable WebViews across 10 applications and found no false positives suggesting that Bifocals's false positive rate is likely below $5 - 10\%$.

We now discuss a few applications and the vulnerabilities we discovered.

Alive. Alive is an application that displays Japanese cartoon images. It has a feature that allows a user to browse for other applications to install. This content is displayed in a WebView, and the landing page and linked pages contain ads. The registered interface provides code to download and install an application. The expected use case is that a user can select an application and click "download" which will download content at a specified URL and save it the SD card. The user is then asked whether they want to install the application. If they accept it, the code launches Android's application installation process.

This introduces multiple risks. One possible attack is that a network attacker or malicious advertisement could save arbitrary files to the SD card, by invoking the registered API with a URL pointing to a site controlled by the attacker. Also, an attacker could trick the user into installing a malicious application, if the attacker launches her attack when the user is browsing an application they are likely to install, or possibly through some other social engineering attack.

The Alive application has two other WebViews with vulnerabilities that allow web content to be downloaded to the internal data folder instead of the SD card.

AIM. The AOL Instant Messaging application contains a vulnerable WebView that accesses the `READ_PHONE_STATE` permission. The application provides an interface to handle successful logins. An attacker (network or web) can use this interface to control the values of the authentication token, session key, screen name, profile URL, and icon URL. This data goes into an "IdentityPreference" data structure which gets used throughout the application, making the application vulnerable to information injection and potentially a CSRF login attack.

Ad Libraries. We also manually examined two of the three ad libraries with potential vulnerabilities: Millennial Media and AdMarvel. The third, Medialets, was obfuscated. Millennial Media and AdMarvel are advertising services that offer rich media ads. Both have registered interfaces that allow the web content to modify the look and feel of the WebView (e.g., view size or layout settings). While neither of these libraries' interfaces invoke protected resources, an attack can still be mounted. An attacker can resize the WebView to take up the whole screen, increasing the chance that the user clicks on it.

Our tool was unable to determine the URLs for these WebViews (due to complexities with URL generation), so we manually confirmed the vulnerability and blacklisted

the two libraries. It is possible that the obfuscated library, Medialets, is also vulnerable, but we conservatively leave that out of our analysis. Only 5 applications use Medialets.

Evaluation of the Tool We find that our tool is able to correctly determine the URL loaded for each WebView in most cases. In both cases, the missing portion of the URL was a value for the URL query string. Ultimately, these query parameters did not affect the landing page, therefore the result from the crawler was correct.

In two cases, the website no longer existed, and in its place were squatter and GoDaddy pages, respectively. Our crawler crawled these pages and found potential vulnerabilities. We believe this to be the correct result as the squatting page would be displayed to the user, making the WebView vulnerable. In fact, this may present a larger threat, as an attacker can easily gain access to the user's application by purchasing the domain.

5.4 Limitations

One limitation of our study is that our data set is two years old. It would be interesting to evaluate Android 4.2 applications. We do not know how the results would differ. (We suspect the results may not change significantly. First, WebViews have increased in popularity, potentially increasing the number of applications exposed to these vulnerabilities. Second, all vulnerabilities still exist in the current platform API. Only one change was made to the JavaScript interface for Android 4.2, which was to require explicit annotations to JavaScript accessible methods (announced on Feb. 14, 2013 [11]). This modification is only applied to applications that set Android 4.2 as the minimum or targeted API. As of Feb. 4, only 1.4% of Android devices operate on Android 4.2 [3], and it is unlikely that many developers have set their applications to restrict distribution to the Android 4.2 platform.)

6 Suggested Improvements

6.1 Current Shortcomings

The core of the excess authorization problem is that *any* content loaded in the WebView is able to invoke application code, making it very easy for developers to unintentionally grant untrusted sources the ability to invoke application code. We conjecture that many of the vulnerabilities we found may be attributable to developer confusion with the WebView system. In particular, we observed three significant pitfalls for developers:

1. WebViewClients transparently change navigation behavior. If a WebViewClient is added, the WebView is implicitly made navigable. A developer who adds a WebViewClient to alter some non-navigation feature will make their WebView navigable, and thus may introduce an excess authorization vulnerability without realizing.
2. We have observed confusion with what the `shouldOverrideURLLoading()` method means and does. Stack Overflow contains many questions on what the method should do [36]. Most commonly, we have observed implementations of the overridden method that load a URL and then return `true`. This is the equivalent of not overriding the method at all or simply returning `false`.
3. A third potential source of confusion is that developers just may not be aware that *everything* loaded in the page or navigated to can invoke the application code.

6.2 Recommendations for Developers

In light of these pitfalls, we suggest ways a developer can reduce their attack exposure:

- **Disable Javascript.** Developers can turn off JavaScript if they do not need it.
- **Restrict navigability.** Developers can restrict the WebView's navigability. This, however, only limits content loaded via links and does not limit content in the document (e.g., frames or JavaScript). Consequently, it is not a complete defense.
- **Limit APIs.** Third, developers can limit the exposure to the API by only registering necessary interfaces. Functionality that should not be made available to web content should be separated out into a different class.
- **Use new Android mechanisms.** Android recently announced a new requirement for accessible interface methods to be annotated with `@JavascriptInterface` for Android 4.2 [11]. Developers should opt in to this by setting the minimum (or targeted) SDK version to Android 4.2. One caveat, however, is that while this may reduce accidental over-inclusion of accessible methods, it does nothing to prevent JavaScript from invoking intentional interface methods. Another caveat is that this approach does not exist for devices running versions older than 4.2. Also, it may take years for Android 4.2 to be used by a majority of phones, and developers may not want to limit their application's user base by targeting 4.2 for a while.

While these do not wholly prevent a vulnerability, they may limit the attack surface.

6.3 Recommendations for the Android Platform

To reduce the risk of unintentional excess authorization, we recommend that the Android platform be modified so that access to an exposed interface is granted only to specified domains instead of all content loaded in a particular WebView. For example, if a WebView loads `foo.com`, only `foo.com` should be allowed to invoke the interface. Other domains should not get access to the interface. Third-party web content loaded via frames should not get access to application code.

Specifically, we propose a policy that limits access by the second-level domain (SLD). The policy maintains a list of allowed SLDs for each WebView, and authorizes all content from such an SLD to invoke any interface registered with that WebView. By default, the list of allowed SLDs is initialized with the SLD of the URL initially loaded in the WebView. If this triggers a redirect, we automatically add the SLD of the target as well. This list can be supplemented by an optional developer-supplied whitelist of acceptable SLDs for each WebView (a WebView-level whitelist).

This approach provides an automated way to secure WebViews, lowering developer burden, while providing flexibility for developers to override the policy if they intentionally want specific third-party content to access the application.

Developer Effort. We evaluated this approach based on the amount of developer effort that would be required to comply with it. We found that 100% of core applications that give access to code are handled automatically by our default policy and do not require any developer effort or other changes.

Effectiveness. Our approach would patch vulnerabilities due to frames and links. It would not patch vulnerabilities due to third-party JavaScript included directly on the landing page as they would obtain the domain of the page.[7]

[7] Our approach also would not mitigate attacks via a XSS vulnerability (which is outside the scope of this work).

We find that of the 18 vulnerable core applications, 11 of the landing pages (61%) would be patched by our proposed policy. The remaining applications load ads directly on the landing page. Our estimate, however, may be an under-approximation of the number of patched pages. Adblock flags actual ads as well as ad providers' JavaScript (such as the Google script that generates the ad). It is possible that the JavaScript subsequently loads the ad content in a frame, in which case our solution would patch the vulnerability; however, this case is not included in our count of patched applications.

7 Related Work

WebViews. We are inspired by the work of Luo et al., which identifies the potential for WebView attacks [29]. They give examples for how webpages can attack applications, how applications can attack webpages, and introduce the excess authorization vulnerability. They perform a brief, primarily manual analysis of the possibility of these vulnerabilities in applications. We extend their work by identifying variations on the basic code exposure attack and enumerating threats from different attackers, including the network attacker and attacks via remote script inclusion and XSS threats. Also, in contrast to their small-scale, manual investigation, we perform a large-scale measurement study and build an automated analysis tool to detect these vulnerabilities.

Saltzman blogged about a WebView-related attack in file-sharing applications [10]. File-sharing applications, such as DropBox, often save files to the application's internal file directory and can be displayed in a WebView. Assuming a malicious file gets saved, this file would then gain access to other files, potentially sending them to the attacker. We present a file-based cross-zone scripting attack that is a more general form of this attack, which can occur in any application. A trusted internal file, as opposed to a malicious file, can load external JavaScript, giving it access to the file system.

Static analysis tools for Android. Researchers have developed static analysis tools to identify other security properties in Android applications. For vulnerability detection, Grace et al. and Felt et al. apply CFG-based static analysis techniques to detect capability leaks across application boundaries [24, 22]. Felt et al. and Au et al. build static analysis tools examine permission overprivilege in Android applications [21, 14]. AdDroid examines overprivilege due to permissions only required by ad libraries [32].

Other static analysis tools focus on the identification of grayware or malicious applications. SCanDroid takes a data-centric approach to reasoning about the consistency of security specifications concerning permissions and databases [23]. Their tool, however, takes Java source code as input. Kim et al. present a bytecode-level static analysis tool to detect privacy leaks. They track location info, IDs (IMEI, IMSI, ICC-ID), audio and video eavesdroppers [26]. Batyuk et al. and Schmidt et al. similarly propose static analysis techniques to identify malicious Android applications [15, 35]. To our knowledge, no tools have been created to analyze Android and web interaction.

In contrast to building static analysis tools from scratch, Scandariato et al. apply the COTS tool, Fortify Source Code Analyzer, to open-source Android applications and use code metrics to infer the likelihood of vulnerabilities [34]. Enck et al. also take advantage of Fortify's SCA but avoid dataset limitations of open source applications by creating a decompiler called *ded* to generate Java source code from an application binary. They examine security properties such as IMEI leakage and resource abuse [19].

8 Conclusion

While WebViews facilitate the creation of rich, interactive applications, they also introduce the potential for attack if developers are not careful. We examine vulnerabilities of WebViews and present Bifocals, which analyzes both Android applications and web content to identify vulnerabilities in applications. We discovered 67 applications that are vulnerable to attack through WebViews.

Excess authorization arises due to a mismatch in authorization expectations. A developer may intend to give code access to a specific website, but in actuality access is granted to anything loaded in the WebView. We propose changes to WebViews to grant code access based on the domain and not the WebView, thereby limiting the opportunity for exposure to malicious JavaScript. Our solution patches 60% of the vulnerabilities we found and requires very little developer effort.

Acknowledgments

This research was supported by Intel through the ISTC for Secure Computing. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of Intel.

References

1. Adblock plus. http://adblockplus.org/.
2. AdMarvel. http://www.admarvel.com/.
3. Dashboards: Platform versions. http://web.archive.org/web/20130205234427/http://developer.android.com/about/dashboards/index.html.
4. HTTP state management mechanism RFC. http://www.rfc-editor.org/rfc/rfc6265.txt.
5. Malware delivered by Yahoo, Fox, Google ads. http://news.cnet.com/8301-27080_3-20000898-245.html.
6. Malware-infected WinRAR distributed through Google AdWords. http://www.zdnet.com/blog/security/malware-infected-winrar-distributed-through-google-adwords/2405.
7. Medialets. http://www.medialets.com/.
8. Millennial media. http://www.millennialmedia.com/.
9. nsIContentPolicy. https://developer.mozilla.org/en-US/docs/XPCOM_Interface_Reference/nsIContentPolicy.
10. Old Habits Die Hard: Cross-Zone Scripting in Dropbox & Google Drive Mobile Apps. http://blog.watchfire.com/wfblog/2012/10/old-habits-die-hard.html.
11. Security Enhancements in Jelly Bean. http://android-developers.blogspot.com/2013/02/security-enhancements-in-jelly-bean.html.
12. Smali and baksmali. http://code.google.com/p/smali/.
13. Times web ads show security breach. http://www.nytimes.com/2009/09/15/technology/internet/15adco.html.
14. AU, K. W. Y., ZHOU, Y. F., HUANG, Z., GILL, P., AND LIE, D. Short paper: a look at smartphone permission models. In *Proc. of the 1st ACM workshop on security and privacy in smartphones and mobile devices* (2011).
15. BATYUK, L., HERPICH, M., CAMTEPE, S. A., RADDATZ, K., SCHMIDT, A., AND ALBAYRAK, S. Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within Android applications. In *Procs. of the 6th International Conference on Malicious and Unwanted Software (MALWARE)* (2011).
16. CHESS, B., AND MCGRAW, G. Static analysis for security. *Security & Privacy, IEEE 2*, 6 (2004), 76–79.

17. CHIN, E., FELT, A. P., GREENWOOD, K., AND WAGNER, D. Analyzing inter-application communication in Android. In *Proc. of the Annual International Conference on Mobile Systems, Applications, and Services* (2011).
18. DI LUCCA, G. A., FASOLINO, A. R., MASTOIANNI, M., AND TRAMONTANA, P. Identifying cross site scripting vulnerabilities in web applications. In *Proc. of the 6th IEEE International Workshop on Web Site Evolution (WSE)* (2004).
19. ENCK, W., OCTEAU, D., MCDANIEL, P., AND CHAUDHURI, S. A study of Android application security. In *Proc. of the 20th USENIX Security Symposium* (August 2011).
20. ENDLER, D. The evolution of cross site scripting attacks. *Whitepaper, iDefense Inc.* (2002).
21. FELT, A. P., CHIN, E., HANNA, S., SONG, D., AND WAGNER, D. Android permissions demystified. In *Proc. of the ACM Conf. on Computer and Communications Security* (2011).
22. FELT, A. P., WANG, H., MOSHCHUK, A., HANNA, S., AND CHIN, E. Permission redelegation: Attacks and defenses. In *Proc. of the 20th USENIX Security Symposium* (2011).
23. FUCHS, A. P., CHAUDHURI, A., AND FOSTER, J. S. SCanDroid: Automated security certification of Android applications. Tech. rep., University of Maryland, 2009.
24. GRACE, M., ZHOU, Y., WANG, Z., AND JIANG, X. Systematic detection of capability leaks in stock Android smartphones. In *Proc. of the 19th Annual Symposium on Network and Distributed System Security* (2012).
25. KANICH, C., CHACHRA, N., MCCOY, D., GRIER, C., WANG, D. Y., MOTOYAMA, M., LEVCHENKO, K., SAVAGE, S., AND VOELKER, G. M. No plan survives contact: experience with cybercrime measurement. In *Proc. of the 4th conference on cyber security experimentation and test* (2011).
26. KIM, J., YOON, Y., YI, K., SHIN, J., AND CENTER, S. ScanDal: Static analyzer for detecting privacy leaks in Android applications. In *Proc. of the MoST* (2012).
27. KIRDA, E., KRUEGEL, C., VIGNA, G., AND JOVANOVIC, N. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *Proc. of the 2006 ACM symposium on Applied computing* (2006).
28. LIVSHITS, V. B., AND LAM, M. S. Finding security vulnerabilities in Java applications with static analysis. In *Proc. of the 14th Conference on USENIX Security Symposium* (2005).
29. LUO, T., HAO, H., DU, W., WANG, Y., AND YIN, H. Attacks on WebView in the Android system. In *Proc. of the 27th Annual Computer Security Applications Conference* (2011).
30. NIKIFORAKIS, N., INVERNIZZI, L., KAPRAVELOS, A., VAN ACKER, S., JOOSEN, W., KRUEGEL, C., PIESSENS, F., AND VIGNA, G. You are what you include: Large-scale evaluation of remote JavaScript inclusions. In *Proc. of the ACM Conference on Computer and Communications Security* (2012).
31. PALLER, G. Dedexer. `http://dedexer.sourceforge.net/`.
32. PEARCE, P., FELT, A. P., NUNEZ, G., AND WAGNER, D. AdDroid: Privilege separation for applications and advertisers in android. In *Proc. of AsiaCCS* (2012).
33. SC MAGAZINE. WhiteHat: 90 percent of websites vulnerable to attack. `http://www.scmagazine.com/whitehat-90-percent-of-websites-vulnerable-to-attack/article/58066/`.
34. SCANDARIATO, R., AND WALDEN, J. Predicting vulnerable classes in an Android application. *Proc. of the 4th international workshop on security measurements and metrics* (2012).
35. SCHMIDT, A.-D., BYE, R., SCHMIDT, H.-G., CLAUSEN, J., KIRAZ, O., YUKSEL, K. A., CAMTEPE, S. A., AND ALBAYRAK, S. Static analysis of executables for collaborative malware detection on Android. In *Proc. of Intl. Conference on Communications (ICC)* (2009).
36. STACK OVERFLOW. Developer sites contradict each other regading webview-shouldoverrideurlloading. `http://stackoverflow.com/q/10865788`.
37. WAGNER, D., FOSTER, J. S., BREWER, E. A., AND AIKEN, A. A first step towards automated detection of buffer overrun vulnerabilities. In *Proc. of Network and Distributed System Security Symposium* (2000).