Security Analysis of Cordova Applications in Google Play

Michiel Willocx, Jan Vossaert, Vincent Naessens
MSEC, imec-DistriNet
firstname.lastname@cs.kuleuven.be

ABSTRACT

Mobile Cross-Platform Tools (CPTs) provide an alternative to native application development that allows mobile app developers to drastically reduce the development time and cost when targeting multiple platforms. They allow sharing a significant part of the application codebase between the implementations for the targeted platforms (e.g. Android, iOS, Windows Phone). Although CPTs provide significant benefits for developers, there can introduce several disadvantages. The CPT software layers and translation steps can impact the security of the produced applications. One of the most well-known and often-used CPTs is Cordova, formerly known as PhoneGap. Cordova has, over the years, taken several steps to reduce the attack surface and introduced several mechanisms that allow developers to increase the security of Cordova applications. This paper gives a statistical overview of the adoption of Cordova security best practices and mechanisms in Cordova applications downloaded from the Google Play Store. For the analysis, over a thousand Cordova application were downloaded. The research shows that the poor adoption of these mechanisms leads to a significant number of insecure Cordova applications.

CCS CONCEPTS

• **Security and privacy** → *Web application security*; *Domain-specific security and privacy architectures*;

KEYWORDS

Cordova, Mobile application development, Security, Google play

ACM Reference format:
Michiel Willocx, Jan Vossaert, Vincent Naessens. 2017. Security Analysis of Cordova Applications in Google Play. In *Proceedings of ARES '17, Reggio Calabria, Italy, August 29-September 01, 2017,* 7 pages.
https://doi.org/10.1145/3098954.3103162

1 INTRODUCTION

Mobile applications have become a must-have for many service providers. They are used to attract new users, and support existing users more efficiently. In order to reach as many users as possible while keeping development and maintenance affordable, an increasing number of developers is using Cross-Platform Tools (CPTs). They allow developers to target multiple platforms (e.g. Android and iOS) using a shared code base. They are, therefore, a promising alternative to native development, where each platform requires dedicated tools and different programming languages (e.g. Objective-C in iOS, Java in Android).

One of the most well-known, and often-used cross-platform tools is Cordova, formerly known as PhoneGap. Cordova is a Web-to-native wrapper, allowing the developer to package Web apps consisting of Web languages such as JavaScript and HTML into stand-alone applications.

Developers have access to a wide range of JavaScript frameworks which aid with tailoring Web applications to mobile device user interfaces. The wide range of development and styling frameworks enables a fast development cycle for applications.

Although cross-platform tools such as Cordova provide many opportunities and advantages for developers, they can also introduce additional security vulnerabilities compared to native applications by introducing additional software layers and/or translation or compilation steps. Since Cordova applications are developed using Web technologies many typical Web attacks such as XSS and code injection [10] can be applied to Cordova applications. In the context of mobile applications, these types of attacks can, potentially, give access to the vast amount of personal information available on the mobile device.

Contribution. The contribution of this paper is twofold. First, over a thousand Cordova applications were downloaded from the Google Play Store. A set of security best practices for Cordova applications were devised and the adoption of these best practices in the downloaded applications were verified. This paper specifically focuses on best practices related to the Cordova platform and does not perform an overall security evaluation of mobile applications. To obtain the required Cordova applications, a tool was developed that can automatically search and download applications from the Google Play Store based on a set of keywords. The tool also automatically decompiles the application and determines with which tool (i.e. native or a specific CPT) the application was developed. The source code of the tool is published online. Second, based on the analysis, a set of guidelines for the development of secure Cordova applications and plugins is presented.

The remainder of this paper is structured as follows. Section 2 points to related work. A general introduction to Cordova and a discussion of the Cordova security metrics used in the evaluation is presented in Section 3. The tool to download and analyze applications from the Google Play Store is presented in Section 4, followed by the results in Section 5. The results are further discussed in Section 6. Finally, conclusions are drawn in Section 7.

2 RELATED WORK

Much research can be found discussing the security of mobile platforms and applications. La Polla et al. [6] give an overview of mobile threats and vulnerabilities, and offer possible solutions.

ARES '17, August 29-September 01, 2017, Reggio Calabria, Italy

ACM ISBN 978-1-4503-5257-4/17/08…$15.00
https://doi.org/10.1145/3098954.3103162

Bhardwaj et al. [1] and Ibtisam et al. [7] present an in-depth security comparison between Android and iOS. These papers provide insight in the general threats to mobile platforms and applications. Other papers focus on specific topics. For instance, Fahl et al. [2] and Onwuzurike et al. [9] assess possible threats and vulnerabilities when setting up TLS connections in Android. Other papers focus on WebView security. For example, [8, 13] and [12] study the impact of Webview-related vulnerabilities on mobile applications and offer mitigation techniques. The results of this research is taken into account when analyzing the security of Cordova applications in this paper.

Several papers focus specifically on security aspects in Cordova applications. Georgiev et al. [3] focus on access control of device resources in Cordova applications. They focus on preventing untrusted code such as in-app advertisements from accessing native device features. Access to these native device features is typically provided by including third-party plugins in the application. By default, these plugins can interact with all the pages in the application. Shehab et al. [11] focus on limiting the access of these third-party plugins by proposing a framework that enables developers to include an access control policy that controls plugin access to the different pages of the application. Jin et al. [5] identify different attack vectors such as contacts, SMSs and barcodes that can be used to inject code in Cordova applications. A mechanism is described that can filter out malicious code from these different attack vectors. The results of these papers are used for the specification of the Cordova security best practices in this paper.

3 CORDOVA APPLICATIONS

The first part of this section describes the architecture of Cordova applications, and provides an overview of the most important components. The second part of this section presents the security best practices used for the security analysis of the Cordova applications from the Google Play Store.

3.1 The Cordova Architecture

A typical Cordova application consists of three important components: the application source, the WebView and plugins (see Figure 1).

Cordova applications are, similar to Web apps, developed in Web languages (i.e. HTML, CSS and JavaScript). Typically, developers use JavaScript frameworks such as Ionic and Sencha which facilitate development of mobile UIs. The application code is loaded in a chromeless WebView. By default, Cordova applications use the WebView bundled with the operating system. An alternative is to include the Crosswalk WebView[1]. The Crosswalk WebView provides uniform behavior and interfaces between different (versions of) operating systems. Hence developers do not need to take into account the differences in behavior/APIs between the WebViews contained in the different (versions of) operating systems.

The JavaScript APIs provided by the WebView do not allow full access to the diverse resources of the mobile device, such as sensors (e.g. accelerometer, gyroscope) and functionality provided by other applications installed on the device (e.g. contacts, maps, Facebook login). Plugins allow JavaScript code to access native APIs

[1]https://crosswalk-project.org

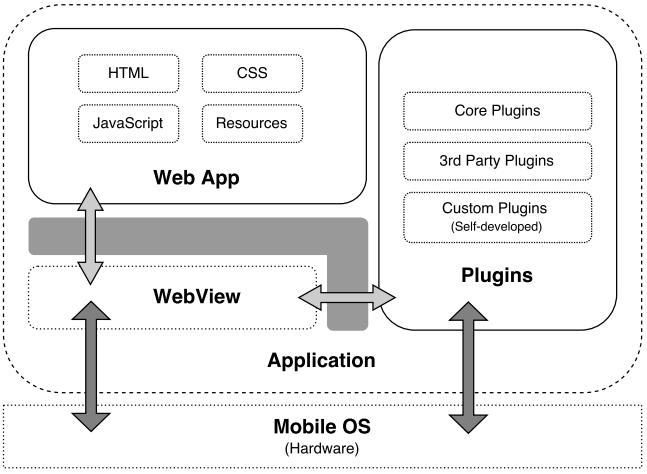

Figure 1: Structure of a Cordova application. Light grey arrows represent JavaScript calls, darker grey represent native calls. The Cordova framework is illustrated by the gray area.

by providing a bridge between the Web code and the underlying operating system. They consist of both JavaScript code and native code (i.e. Java in Android, Objective-C in iOS). The JavaScript code provides the plugin's interface to the developer. The native source code implements the functionality of the plugin and is compiled when building the application. Commonly used functionality like GPS are offered by Cordova as *core plugins*. Further functionality is provided by over 1000 third-party plugins which are freely available in the Cordova plugin store[2].

3.2 Security Best Practices

Cordova provides several mechanisms and developer guidelines[3] to enhance the security of Cordova applications. This section discusses the set of criteria used for the security evaluation of Cordova applications in the Google Play Store.

- **Cordova version.**
  Cordova/PhoneGap already exists for over five years. Over all these years, several new security mechanisms have been added and security critical bugs were fixed. The Cordova website provides write-ups of all performed security patches. For instance, the release of Cordova 3.5.1[4], released in August 2014 contained several important bug fixes. One of the problems fixed in this release was a vulnerability that allowed cross-application scripting via Android intent URLs. Cordova applications can be launched through an intent. Attackers could cause Cordova applications to start up with a different start page than the developer intended, including other HTML content stored on the Android device by crafting a special intent URL.

[2]https://cordova.apache.org/plugins/
[3]A brief overview of this can be found in the Cordova documentation: https://cordova.apache.org/docs/en/latest/guide/appdev/security/
[4]http://cordova.apache.org/announcements/2014/08/04/android-351.html

- **The whitelisting mechanism**
  The Cordova whitelist allows three types of whitelist definitions: `access origin`, `allow-intent` and `allow-navigation`. The three types of whitelists are further discussed below. The analysis gives an overview of how the different whitelisting mechanisms are used in practice.
  The `access-origin` whitelist allows the developer to control to which domains the application is allowed to make network requests. If no access tag is defined in the whitelist, applications only have access to resources on the device. This is specifically relevant when external content is loaded in an iFrame. Developers need to be aware that loading code in iFrames does not provide the same security properties as they do in traditional Web apps. Code running in iFrames has the same privileges and access to native APIs as regular application code. By default, the whitelist contains a wildcard, allowing access to all domains. Security best practices recommend removing this wildcard and explicitly whitelisting the necessary HTTPS domains. Applications that use the wildcard to allow all domains or whitelist HTTP domains could be vulnerable to code injection and MiTM (Man-in-The-Middle) attacks.
  The `allow-navigation` whitelist controls which URLs the WebView can navigate to. By default, no navigation whitelist is defined, only allowing navigation to `file://` URLs. Most Cordova applications only need the default configuration because application files are included in the APK file. Hence, in this case the developer should not change anything in the whitelist. If, however, WebView navigation is required in an application, the whitelisted URL should always use the HTTPS protocol. It is, further, advised to only navigate the WebView to content managed by the developer, as loaded code will gain complete access to all APIs available to the the application.
  The `allow-intent` whitelist controls which URLs the app is allowed to ask the mobile platform to open. By default, several `allow-intent` items are included, such as opening links to websites in the browser, redirecting a phone number to the system's SMS or phone call application, opening the app store and opening an address in a navigation application. The default whitelisted intents provide specific, often used, functionality. It is, however, not advised to replace the default whitelist by a wildcard, as it can potentially be abused to trigger unintended functionality on the mobile device.
- **Content Security Policy (CSP[5])**
  Cordova recommends including CSP tags in the application's HTML page(s). It protects against common Web-based security threats such as XSS (Cross-Site Scripting). CSP allows a developer to define trusted sources for each content type (e.g. scripts, style). Content from sources not explicitly listed cannot be loaded in the application. A CSP definition, by default, also disables the use of `eval()` and the execution of inline JavaScript, which are both commonly used for XSS attacks. It is a complementary layer of security with the whitelisting mechanism described above. The whitelist is not able to filter all types of requests. For example, `<video>` tags and WebSockets are not blocked. Using the whitelist is recommended since it provides protection when the application is run on older WebViews that do not yet support CSP. Table 1 gives an overview of the CSP support in Webviews. Even though the use of `eval()` discouraged, it is a commonly used feature of the JavaScript language and many frameworks (e.g. JQuery) rely on it. Therefore, CSP tags (e.g. `'unsafe-inline'` or `'unsafe-eval'`) can be added to re-enable specific functionality. Regular Web apps load the JavaScript framework scripts form an online source. This allows the developer to set a CSP rule that only allows `eval()` for that specific source. However, in the context of a Cordova application, this is not possible. Applications typically need to be available offline, which means that the JavaScript framework files need to be included in the application. CSP's granularity does not allow whitelisting specific files. Hence, developers that use frameworks relying on `eval()` are forced to allow `unsafe-eval` for the whole application (i.e. `self` source).

Webview	Minimum version
Android	Andoid 4.4
iOS	iOS 9
Crosswalk	Crosswalk 5

Table 1: Webview CSP compatibility

- **Minimal Android version**
  For some security features, Cordova relies on the underlying operating system to provide the necessary functionality. For instance, in Android Gingerbread and older, the whitelisting mechanism does not work. Hence, the `min-target-sdk` parameter in the AndroidManifest should be above 10.

4 APPLICATION DOWNLOAD AND ANALYSIS TOOL

This section discusses the tool[6] that searches for applications in the Google Play Store, downloads the APK file of found applications and then runs an analysis on the decompiled file. The results of the analysis is stored in a database. This section further discusses the implementation of the tool and goes into more detail on how the security best practices discussed in the previous section are checked.

For several components of the tool, website automation is required. This is implemented using Selenium[7] in combination with a WebDriver like Firefox or PhantomJS, depending on the required functionality. PhantomJS is headless and therefore faster. Firefox provides more functionality such as handling native dialog boxes, which is a requirement to automate a download procedure.

4.1 Finding Applications

The crawler is used to search the Android Play Store for applications based on a list of keywords. It stores the package name of each

[5]More information can be found on https://content-security-policy.com.

[6]Source code available at https://github.com/MichielWillocx/CodovaAnalysisStore.
[7]http://www.seleniumhq.org

found application along with other available information such as the application's rating, the amount of downloads and the date of the last update. The crawler was implemented using website automation of the Google Play Store.

4.2 Downloading applications

Downloading applications is the most time consuming task. Two methods were used to download applications: automation of several third party websites, and automation of the Google Play Store.

4.2.1 Automation of third party websites. Several websites such as *APKPure* and *apk-dl*[8] provide services to download the APK files of free Android applications. Using Selenium, it is possible to automate these websites. This strategy, however, has several disadvantages. First, this method is very slow. Further, some downloads fail when following this procedure because some apps are only available in certain countries, and servers can be hosted anywhere. Moreover, most websites have mechanisms which restrict the amount of requests a user can do in a certain time period, resulting in time-outs and IP bans. In total, a little over 25% of download attempts via this strategy failed.

4.2.2 Automation of the Google Play Store. To download the applications which failed with the above procedure, the Google Play Store itself was automated. The Play Store allows users to log in with a Google account and push apps to devices linked to their Google account. As soon as the user decides to install an app from the browser, it starts downloading and installing on the device. The *Android Debug Bridge (ADB)* is used to pull the application from the device once installed. This method is quicker, more error-resistant, and did not trigger any kind of bans.

4.3 Decompiling applications

Before an application can be analyzed, a decompilation step is required to extract the content of the APK file. *APKTool*[9] provides this functionality. It restores the AndroidManifest file along with other resources and decompiles the native code into *Smali* files. For preserving storage capacity, the tool removes, the original APKs after storing the decompiled version of the application.

4.4 Analyzing applications

Before performing a Cordova-specific analysis, a preliminary analysis is required to filter the Cordova applications. CPT usage of an application can be checked by analyzing the *AndroidManifest.xml* file. Most CPTs insert specific components in the manifest, based on which the used CPT can be determined. For instance, the Cordova platform includes its custom WebView activity in the manifest. Furthermore, the *AndoidManifest.xml* file also contains the Android permissions required by the application.

The typical file structure of a decompiled Cordova application is illustrated simplified in Figure 2. For the analysis of the Corodva application, the contents of two folders in the APK is important: the */www* folder and the */res* folder. The */www* contains several important files. The *index.html* file is loaded into the WebView when the application launches. If the application contains a CSP policy, it is found in this file. This folder also contains a */plugins* folder, which contains the JavaScript code of every plugin used in the application. Finally, any used JavaScript frameworks such as Ionic and Sencha can also be found in this directory. The */res* folder contains the Cordova *config.xml* file. This file contains all Cordova-related configurations such as the used plugins, the whitelist settings and other preferences a developer can define.

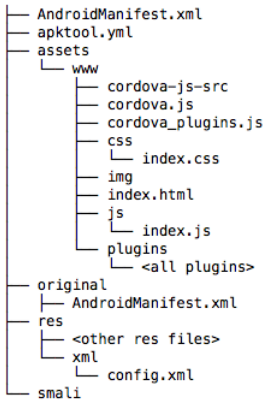

Figure 2: File structure of a decompiled Cordova application.

5 RESULTS

For this research, 250 random keywords were used to find applications. With these keywords, 46.000 applications were found, of which 43.000 were downloaded, decompiled and analyzed. Together, all decompiled applications take 3TB of disk space. Of those 43.000 applications, 1192 were Cordova applications. This section gives an overview of the adoption of Cordova security best practices in those applications.

5.1 Cordova version.

Over 40 different versions of Cordova were found in the analyzed applications. The distribution of Cordova versions is presented in Table 2, along with their release date. The table shows that a significant part of the used Cordova versions is outdated. More than half of the applications use a Cordova version of over two years old at the moment of writing. For comparison, Table 3 gives an overview when applications were last updated. Although almost 40% of the applications received an update in 2016, only 20% was using the last version of Cordova which was released in 2015. This indicates that many developers do not update the version of Cordova used during development of the application when pushing updates.

5.2 Whitelist

Table 4 gives an overview of the use of the different whitelisting mechanisms in the analyzed applications. The first column contains the type of whitelist, the second column shows the amount of applications that contain that whitelist tag, the third column contains the number of entries of that specific tag in all applications and the last column displays the number of applications that contain a secure configuration for that type of whitelist.

[8] https://apkpure.com/ and https://apk-dl.com/
[9] https://ibotpeaches.github.io/Apktool/

Cordova version	Release dates	%Apps
Now - 5.0.0	Nov. 2015	20,60
5.0.0 - 4.0.0	Apr. 2015	15,57
4.0.0 - 3.5.0	May 2014	51,62
3.5.0 - 3.0.0	Oct. 2013	6,60
3.0.0 - ...	Older	8,62

Table 2: Distribution of Cordova versions.

last update	# Apps	% Apps
before 2013	62	4,18
in 2013	53	3,57
in 2014	326	21,98
in 2015	507	34,19
in 2016	535	36,08

Table 3: Dates of last update.

	# apps	# entries	# secure
Allow navigation	141	212	4
Access origin	1187	2627	2 (+5)
Allow intent	260	1392	-

Table 4: Whitelist overview.

Allow-navigation whitelist. Although the application's config file does not contain a navigation whitelist entry by default, 141 (11,8%) of the examined applications contain this entry. Only four applications have only HTTPS domains whitelisted. 107 apps allow all domains (*), and 30 contain at least one HTTP domain. A more secure alternative for opening external URLs in a WebView is to use the *InAppBrowser* plugin. This is used by many applications, our research found 891 of the analyzed applications use this plugin, which makes this the second most used plugin overall.

Access-origin whitelist. The default whitelist allows access to all domains (*). Nearly all examined applications contain one or more access-origin whitelist entries. In many cases, developers add new entries to the whitelist, but preserve the wildcard entry. Hence, 1067 applications contain (at least) the wildcard enrty. Furthermore, of the remaining applications 118 contain at least one entry with a HTTP domain. Only two applications only whitelist HTTPS domains. Finally, five applications contain no access-origin whitelist, which is also a secure option. In total, seven applications contain a secure access-origin whitelist configuration.

Allow-intent whitelist. The allow-intent whitelist is the most recent type of whitelist. Only 260 (22%) of the applications contain this type of whitelist. Of these applications, 128 (49%) keep the default values. 48 (18%) allow all possible intents (*).

5.3 CSP

Table 5 shows the analysis results of the apps containing a CSP definition. For each content type, the total number declarations (decl.), `unsafe-eval`, `unsafe-inline`, HTTP domains and wildcards (*) is listed. 133 apps containing a CSP policy were found. With respect to security, the most important content type is `script-src`. Most applications that contain a `script-src` definition allow the use of `unsafe-inline` and `unsafe-eval`. Furthermore, 43 applications allow loadings scripts from all domains (*). When a content type is not specified in a CSP policy, the `default-src` CSP is applied. 84 applications allow loading content from all domains for the default content type.

Content type	#decl.	unsafe eval	unsafe inline	HTTP domain	*
`default-src`	109	24	7	3	84
`connect-src`	29	1	11	3	18
`script-src`	92	85	84	3	43
`style-src`	107	1	103	3	41

Table 5: CSP use

Table 6 shows the amount of `eval()` method calls used by the apps. Of the tested applications, 38% does not contain any `eval()` functions at all. A quarter of the applications contains more than 10. For this parameter, we checked for `eval()` function calls in the complete application code, which also contains JavaScript frameworks and possible plugins. Both JavaScript frameworks and plugins are known to include `eval()` calls in several occasions and do, therefore, contribute to the total. For example, JQuery, which was the most found JavaScript framework in this research (see Discussion), contains several `eval()` methods.

# eval method calls	# apps	% apps
No evals	566	38.14
0 to 10	560	37.74
10 to 50	218	14.69
More than 50	140	9.43
Max # evals found in 1 app		2452

Table 6: amount of eval-occurrences in applications

5.4 Minimal required Android version

Table 7 gives an overview of the minimal required android version required by the Cordova applications. The table shows that 60% of the applications can run on Android 2.3 (Gingerbread) and lower. Running these applications on these old Android versions could potentially result in unsafe behavior. However, according to the Android website[10], only 1% of the devices still uses these outdated Android versions.

[10] https://developer.android.com/about/dashboards/index.html

# Android version	# apps	% apps
1.x and 2.x	875	59.81
3.x	23	1.57
4.x	563	38.48
5.x	2	0,14

Table 7: Minimal required Android version to run the application

6 DISCUSSION

This section consists of two parts. The first part presents some interesting general statistics of the applications downloaded from the store. The second part discusses some of the pitfalls of Cordova application that lead to some of the commonly found insecure configurations. Further, some less technical security best practices are discussed.

6.1 Market Analysis

6.1.1 CPT Market Share. The CPT landscape is very diverse. Many different tools exist that rely on different technologies. Cordova is one of the most popular tools, as shown in Table 8. This table shows the distribution of cross-platform tools in the analyzed applications, along with the average application rating (max score of five) and the percentage of these applications that are categorized as games. "Other" indicates that the application is either native or developed with a cross-platform tool not included in this analysis. Over a quarter of the tested applications is developed using one of the considdered cross-platform tools. The most used tool was Unity, of which most found applications were in the Game category.

Cordova is the second most used cross-platform tool with 3,45% of the analysed applications. The table also shows that the average app ratings is fairly independed of the tool used for development (either native or cross-platform).

	%Apps	%Game	Rating
Unity	19,63	93,10	4,30
Cordova	3,45	9,78	4,13
Adobe Air	3,04	72,86	4,17
Xamarin	0,63	9,52	3,99
Titanium	0,47	3,43	3,83
React Native	0,10	4,76	4,13
Other	72,68	24,33	4,33

Table 8: Distribution of found applications.

6.1.2 JavaScript Framework Market Share. Table 9 shows the distribution of the JavaScript frameworks used in the analysed Cordova applications. For this test, a list of over 20 commonly used JavaScript Frameworks was compiled and searched for in the decompiled applications. The nine most commonly used frameworks are explicitly included in the table, the others are grouped in `Other`. An application can use more than one JavaScript framework. Out of the 1483 analysed Cordova applications, 777 (52%) uses at least one JavaScript framework. The table shows that jQuery is the most commonly used JavaScript framework. Angular is the second most used JavaScript framework. In the setting of mobile applications, it is often used in combination with Ionic, which is the third most commonly use framework.

JS framework	# apps	%Apps
jQuery	656	49,96
Angular	201	15,13
Ionic	167	12,72
Bootstrap	163	12,41
Onsen	50	3,81
Sencha	25	1,90
Intel XDK	18	1,37
Kendo	12	0,91
Zepto	7	0,53
Other	14	1,07

Table 9: Usage of JavaScript frameworks.

6.1.3 Cordova Plugin Use. Table 10 shows the distribution of Cordova plugins from the Cordova market place found in the analysed applications. This table shows that a set of 155 plugins from the Cordova plugin store accounts for the marge majority of plugins in Cordova applications. 502 plugins found in this research did not originate from the Cordova plugin store. On average, each of these *unofficial* plugins is only used 8 times. In many cases, developers create a set of custom plugins, which are used in Cordova applications created by that (team of) developer(s). On average, each application contains 14 plugins. The most plugins found in one application was 14.

	# plugin usages	# unique plugins
From plugin store	12479	155
Not from plugin store	4027	502
Total	16506	657

Table 10: Overview of the plugins used in the Cordova applications.

Plugins provide additional interfaces to the Web application that can be used to, for instance, access device resources not available with the HTML5 APIs provided by the WebView. Therefore, using plugins in an application introduces several risks. First, additional device resources can be accessed via the plugin if malicious code is injected in the WebView. Adding plugins typically also automatically adds a set of permissions, required by the plugin to function correctly; to the application. Hence, minimizing the number of plugins in the application is important to reduce the impact of potential

attacks. Plugins can also contain vulnerabilities that could be exploited to attack the application. For instance, some application update plugins do not adequately authenticate the application update, allowing attackers to inject malicious code [4]. Hence, before including specific plugins, it is recommended to assess the trustworthiness of the plugin, especially if it provides security critical functionality such as application updates.

6.2 Developer Guidelines

The results of the analysis presented in the previous section demonstrate that many Cordova applications found in the app store do not adequately adopt security mechanisms and best practices developed and proposed by the Cordova community. Based on our analysis, we discuss some common pitfalls and present some general guidelines and rules of thumb for application developers, plugin developers and Cordova developers.

6.2.1 App developers. App developers need to be aware that the Cordova version used in the application is not automatically updated. In order to keep working with the most recent (and secure) version, Cordova has to be manually updated for each application separately.

In most Cordova applications, plugins are required to access specific devices features. In most cases, the required functionality is available in the Cordova plugin store. Developers should, however, carefully select the right plugin. Even though several factors such as the amount of monthly downloads and the activity and issues on the Github page can be good indicators of the quality of the plugin, developers are advised to always check the source code of a plugin before including it in an application. This also includes checking for unnecessary Android system permissions. Similar to the Cordova framework itself, the plugins receive no automatic updates. Hence, developers should regularly check if updates are available for the used plugins and perform the required updates manually.

Last, because of the way Cordova applications work (i.e. at runtime interpretation of uncompiled JavaScript code), they are easily reverse engineered. The JavaScript code can be revealed by unzipping the APK file. Hence, developers should always assume that attackers have access to the source code.

6.2.2 Plugin developers. Plugin developers have the responsibility to provide secure and developer-friendly software components for app developers. Plugins with a comprehensive API and an extensive documentation accompanied by some code samples, are more likely to gain traction in the community. For security functionality, it is especially important that plugins use the platform-specific APIs provided by the native platforms. Further, plugins need to be kept up-to-date in order to make use of the newest platform's features and the amount of required permissions should be limited to the strict minimum.

6.2.3 Cordova. Developers of the Cordova platform can help increase the security of applications made with the Cordova framework in multiple ways. First, some security settings could have better defaults, especially the whitelist. Wildcards should never be the default value. Second, in the most recent Cordova versions, for each new project, a standard CSP added to the `index.html` file. This CSP does, however, not include the most secure options. A more secure option would be to not allow inline JavaScript or `eval()` to ensure that the default setup is the most secure setup. Developers that wish to diverge from this have to manually adjust this default. Last, in order to keep versions of plugins and the Cordova framework itself up-to-date, a mechanism that automatically notifies the developer of updates could be implemented.

7 CONCLUSIONS

This paper presented an analysis of the use of security best practices in over 1000 Cordova applications found in the Google Play Store. Even though the Cordova framework introduces several Web-based attack vectors compared to native applications, several straight-forward mechanisms can fix the most common issues. A list of relevant security parameters was compiled and checked for each of the found applications. The results show that, for several important parameters, poorly chosen defaults along with carelessness/unawareness of developers leads to a significant number of Cordova applications with insecure configurations in the Google Play Store.

REFERENCES

[1] Akshay Bhardwaj, Kailash Pandey, and Roopam Chopra. 2016. Android and iOS Security-An Analysis and Comparison Report. *Int'l J. Info. Sec. & Cybercrime* 5 (2016), 30.

[2] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. 2012. 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 '12)*. ACM, New York, NY, USA, 50–61.

[3] Martin Georgiev, Suman Jana, and Vitaly Shmatikov. 2014. Breaking and fixing origin-based access control in hybrid web/mobile application frameworks. In *NDSS symposium*, Vol. 2014. NIH Public Access, 1.

[4] Cristiano Inacio Lemes, Michiel Willocx, Vincent Naessens, and Marco Viera. 2017. An Analysis of Mobile Application Update Strategies via Cordova. (2017).

[5] Xing Jin, Xuchao Hu, Kailiang Ying, Wenliang Du, Heng Yin, and Gautam Nagesh Peri. 2014. Code Injection Attacks on HTML5-based Mobile Apps: Characterization, Detection and Mitigation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, New York, NY, USA, 66–77.

[6] Mariantonietta La Polla, Fabio Martinelli, and Daniele Sgandurra. 2013. A survey on security for mobile devices. *IEEE communications surveys & tutorials* 15, 1 (2013), 446–471.

[7] Ibtisam Mohamed and Dhiren Patel. 2015. Android vs iOS security: A comparative study. In *Information Technology-New Generations (ITNG), 2015 12th International Conference on*. IEEE, 725–730.

[8] Patrick Mutchler, Adam Doupé, John Mitchell, Chris Kruegel, and Giovanni Vigna. 2015. A large-scale study of mobile web app security. *Mobile Security Techologies* (2015).

[9] Lucky Onwuzurike and Emiliano De Cristofaro. 2015. Danger is My Middle Name: Experimenting with SSL Vulnerabilities in Android Apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec '15)*. ACM, New York, NY, USA, Article 15, 6 pages.

[10] Top OWASP. 2013. Top 10–2013. *The Ten Most Critical Web Application Security Risks* (2013).

[11] Mohamed Shehab and Abeer AlJarrah. 2014. Reducing Attack Surface on Cordova-based Hybrid Mobile Apps. In *Proceedings of the 2Nd International Workshop on Mobile Development Lifecycle (MobileDeLi '14)*. ACM, New York, NY, USA, 1–8.

[12] Tom Sutcliffe and Adrian Taylor. 2015. The Lifetime of Android API Vulnerabilities: Case Study on the JavaScript-to-Java Interface. In *Security Protocols XXIII: 23rd International Workshop, Cambridge, UK, March 31-April 2, 2015, Revised Selected Papers*, Vol. 9379. Springer, 126.

[13] J. Yu and T. Yamauchi. 2013. Access Control to Prevent Attacks Exploiting Vulnerabilities of WebView in Android OS. In *2013 IEEE 10th International Conference on High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing*. 1628–1633. https://doi.org/10.1109/HPCC.and.EUC.2013.229