An Analysis of Mobile Application Update Strategies via Cordova

Cristiano Inácio Lemes, Michiel Willocx and Vincent Naessens
Faculty of Engineering Technology, MSEC, imec-DistriNet,
KU Leuven, Technology Campus Ghent,
Gebroeders Desmetstraat 1, 9000 Ghent, Belgium
{inaciolemes.lemes,*surname.name*}@kuleuven.be

Marco Vieira
CISUC – Centre for Informatics and Systems
FCTUC – University of Coimbra
3030-290 Coimbra, Portugal
mvieira@dei.uc.pt

***Abstract*—The demand for mobile apps is increasing steadily. To maximize revenue in Business to Customer (B2C) settings, multiple platforms and devices must be supported, which leads to increased development cost. Mobile App Cross Platform Tools (CPTs) tackle this problem as they allow to deploy and run a single codebase on multiple platforms. Cordova is a popular framework for cross-platform development. Multiple plugins support often recurring concerns. One prototypical example is update plugins. This paper focuses on the assessment of update plugins in Cordova, supporting the distribution of code updates. The paper evaluates the most commonly employed ones and compares them against traditional version updates with respect to security and relevant quality parameters. We show that improvident plugin selection and bad developer practices may seriously undermine the security and quality of the mobile apps.**

Keywords–Cross-Platform Tools; Security.

I. INTRODUCTION

The power of mobile devices such as smartphones and tablets has begun to rival personal computers over the last decade. The number of available devices has been increasing and, simultaneously, the amount of apps available in app stores has grown significantly, ranging from business-critical (e.g., banking) to social media [1], [2], [3], [4]. From the operating systems currently available on the market, two players have a substantial market share: Android and iOS. App developers must at least support those platforms to reach a large number of end users, although this fragmentation places a significant burden on the overall development cost. In practice, app development companies must acquire expertise in both iOS and Android development, and the development cycle must be undertaken for two distinct platforms.

Mobile App Cross-Platform Tools (CPTs) allow to deploy a single codebase on multiple platforms, and a wide variety of CPTs exist today [5]. Cordova [6] – formerly known as Phonegap – is very popular and applies web-to-native technology. This CPT enables the development of mobile apps using JavaScript, HTML5 and CSS3 instead of using the native development language [7], [8]. The app code is wrapped into a stand-alone application integrated with a Webview. Cordova offers a set of plugins for accessing device sensors like cameras, calendar, and GPS. Each plugin is split into two parts: (i) the native code which accesses the device feature and (ii) the web code that creates an interface between the application and the native code.

Update plugins are frequently used in Cordova, as they support updating application code without having to upload a new version to the platform's app store (this strategy only enables the updating of webcode, as native code cannot be changed via any channel except the app store). Although multiple update plugins exist and may therefore be integrated within a Cordova app, different approaches are employed internally to support web code updates.

This paper assesses and compares the major update plugins considering their functional behaviour and impact on qualitative properties. We selected major security, user experience, and performance criteria of the Cordova update plugins based on in-depth interaction with Small and medium-sized enterprises (SMEs) [9] The selection of the most appropriate plugin is demonstrated as being crucial to increase the overall app quality and that not all plugins are equally trustworthy. For example, a bad selection strategy can undermine an app's overall security. Finally, the paper compares the Cordova update strategies against more traditional Android version updates. This paper shows that hot code updates offer a viable solution for minor updates and bug fixes to applications. However, large version updates that introduce new permissions or add new plugins require the user to go through the OS supplied application update process. Moreover, using plugins for remote code updates can potentially introduce additional vulnerabilities. Note that the scope of this work is updating strategies on the Android platform. Android was selected for this research because many update plugins only have Android support. Nevertheless, most ideas presented in this work also apply for iOS, as described in Section VI.

The remainder of this paper is structured as follows. Section II discusses background and related work. A classification of code update strategies in the context of mobile apps in addition to an overview of Cordova plugins selected for this study are detailed in Section III. Section IV lists the criteria used when assessing the update strategies. The evaluation of each strategy and a comparison among all strategies is provided in Section V. Section VI reflects on the results. Finally, conclusions are drawn in Section VII.

II. BACKGROUND AND RELATED WORK

Web apps allow servers to host their code and clients can launch the code in their browser by typing in the correct URL. This is a straightforward means of offering content or services to smartphone or tablet users by way of a single codebase. The code consists of web technologies like HTML, CSS and Javascript and may be easily updated without app store intervention. Also, end users are not required to install

 ISBN: 978-1-61208-568-5

an additional app. Although this approach is highly flexible, it is also associated with drawbacks. First, the availability and responsiveness of the application is dependent upon the Internet connection of the mobile device. Second, mobile browsers do not support advanced access to hardware sensors and data, which constrains functionality and negatively impacts user experience.

Hybrid approaches [10] are applied in many CPTs and aim at combining the advantages of web technologies and native functionalities. This requires a native web container embedded in the application to allow web code to be executed on multiple platforms. Native capabilities are accessible via a Javascript bridge and native code execution enables access to device sensors. A hybrid application must be included in the app store and subsequently installed on the user's device. Cordova is the most popular hybrid approach available nowadays [7].

Recent research has focused on assessing and comparing different CPTs in both a qualitative and quantitative way. Heitkötter et al. [6] evaluate four CPT strategies and compare them against native app development. Their work focuses on two major characteristics: *infrastructural support* and *development*. The latter covers an analysis of all development cycle steps. Assessments are completed using a scale from 1 (*very good*) to 6 (*very poor*). Rieger and Majchrzak [11] build further on this work and propose extensions and revisions for evaluations CPTs. They compared two CPTs and applied their assessment to multiple devices. The evaluation criteria was split into four groups: *infrastructure*, *development*, *app*, and *usage*. A *weight* was also assigned to each criterion for each CPT. A detailed evaluation of CPT performance was realized by Willocx et al. [12]. Ten CPTs were assessed and compared against native development using multiple performance criteria. The assessment was performed on multiple iOS, Android and Windows Phone devices. Other studies focus on selecting the most feasible CPT for a specific application or set of applications [7], [8], [13]. All these contributions focus on usability and performance. However, update strategies in particular and plugins in general are not evaluated from a research perspective.

Many mobile applications rely on the Internet to download or upload content. This makes them potentially vulnerable to both passive and active attacks [14]. De Ryck et al. [15] analyze such network attacks. Vashisht et al. [16] propose splitting mobile threats into three categories: *application-based threats* are vulnerabilities concerning applications installed on the device; *web-based threats* expose security issues in the mobile browser and applications which download content from the Internet; and *network-based threats* originate from the mobile or local wireless network. In practice, mobile applications that access the Internet are potentially vulnerable to any of these threats.

Other research focuses on assessing the security of major mobile operating systems. La Polla et al. [17] offer an overview of mobile threats and vulnerabilities before presenting possible solutions to such threats. Peijnenburg [18] studied security considerations concerning Android. Bhardwaj et al. [19] present an in-depth security comparison between Android and iOS.

This paper evaluates and compares alternative *Update Strategies* for mobile applications developed with *PhoneGap/Cordova*. It focuses on the strengths and constraints related to security, user experience and performance of multiple Cordova update plugins. Moreover, we compare our findings to traditional version updates in Android.

III. Update Strategies Overview

Mobile apps may be upgraded after they are published in an app store. Multiple reasons can trigger code modifications, including the addition of new features, the modification of graphical user interface, and fixing defects, for example. In practice, code update strategies can be classified according to three categories: *installing* a new version, *storing* new mobile code in a client side dedicated folder, and *loading* updated code from a web server. These approaches are discussed below in greater detail. Our analysis focuses on plugins that are available at the official Cordova plugin store [20].

a) Installing updates: The developer submits a new installation file to a server and the end users may upgrade the app by installing this file on their devices.

- *Google Play* [21] is the default store for Android applications, but other marketplaces exist [4]. APK files are submitted to the store and published shortly after human revision has taken place.
- Although the Cordova framework does not support modifications in the native code of the application, a dedicated plugin *Cordova-plugin-app-update* enables updating the application by executing an installation file. The installation file can reside at any server selected by the developer.

b) Storing modified web code: The developer submits a set of files containing app code to a server, after which the app may download and store them in the device. Cordova uses a dedicated *www* folder to store, amongst others, business logic and user views. New, updated web code, cannot be stored in this folder because it is read-only. Hence, update plugins store updates outside of the applications. Four frequently used plugins apply this strategy:

- The most frequently downloaded Cordova update plugin according to the plugin store is *cordova-plugin-meteor-webapp*. This is a new version of *meteor-cordova-update-plugin* that fixes some bugs while introducing new features such as performing a rollback to the previous version when the new version is unstable. Any server can host code updates.
- *Cordova-plugin-code-push* is developed by Microsoft and is the second most frequently downloaded update plugin belonging to this category. Each new app release is stored on a server hosted by Microsoft. *CodePush CLI* [22] is a tool that helps developers managing app updates.
- The *cordova-hot-code-push-plugin* is the third most frequently downloaded plugin. Updates are hosted on an Amazon server. *Cordova Hot Code Push CLI* [23] assists developers on managing new releases of the application.
- Finally, *cordova-plugin-dynamic-update* does not employ any mandatory server, so developers can host the update on any server. This plugin only implements the most basic functionalities i.e. downloading new content from the Internet and pointing the WebView to the new files.

 ISBN: 978-1-61208-568-5

c) Loading remote code updates at runtime: This strategy downloads remote code each time the app is running. The user launches the application using a browser that connects to the server from which the application is loaded.

IV. Evaluation Criteria

Two major stakeholders may be identified in the context of the update process, namely the *developer* and the *end user*. The developer is responsible for upgrading a mobile app and submitting the upgrade to the server. The end user retrieves new releases when they are available on the server and integrates it in the mobile app. At least two components (or hardware platforms) are involved in a upgrade operation: the *server* and the *mobile*. The server stores new application releases sent by the developer, after which they may be downloaded to a mobile device. This is based on two main operations: *submit* and *retrieve*. In practice, developers submit new app code to the server, whereas end users retrieve the update some time afterwards. Both the server and the mobile device need Internet access to perform these operations.

This section lists the set of properties that are used to compare alternative update strategies. It resulted from compilation of feedback offered by mobile app developers in the scope of a technology transfer project CrossMoS [9].They are split in two groups: *Security Criteria* and *Quality Criteria*. The former concerns the security aspect of the update process, while the latter focuses on user experience and performance.

A. Security Criteria

Figure 1 shows the interaction between the stakeholders and the components involved in the update process, as well as security concerns and their relation to the update process.

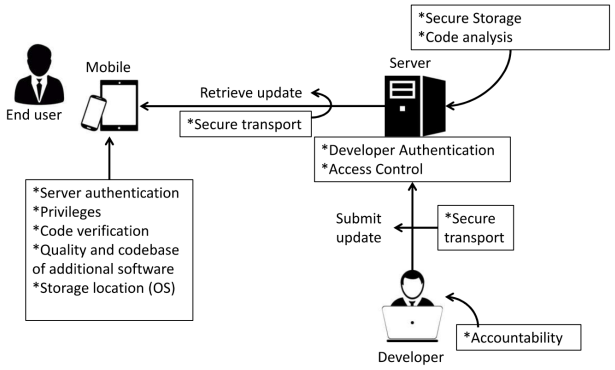

Figure 1: Security Parameters

Secure transport is a major security concern during both the submission and retrieval phase. Inappropriate communication properties can compromise mobile app security and even the device itself after the upgrade. An attacker should not be able to modify the code in transit, typically attempted by Man-In-the-Middle (MiTM) attacks (see details in the Section VI).

Developer authentication is an important security concern during the submission phase. It discourages misbehaving entities from uploading malicious code. Depending on the specific authentication strategy used, multiple updates may be linked to the same (pseudonymous) user, or even linked to an identifiable person or organization.

Accountability goes one step further than authentication and requires a stakeholder to be held accountable by a dispute handler (such as a law enforcement entity) for malicious behaviour. This study focus particularly on the accountability of developers themselves. More specifically, we evaluate whether developers can be held accountable for the submission of malicious code to the server. Appropriate authentication can already identify the individual behind a malicious submission, but only signed code can be probably linked to a physical entity.

Access control measures are implemented on the server side and restrict the possible actions that may be taken by stakeholders when accessing the server. Only authenticated developers and, maybe, collaborators or employees within the same organization, can modify application code (potentially with different access restrictions).

Code analysis/verification can be performed by the update server to check for malicious behaviour in the submitted code. Malicious code potentially exploits privileges (or permissions) already granted to apps and likely leaks sensitive personal data stored in the mobile device to an attacker. Moreover, it can disturb the correct functioning by draining the battery or stealing money in the background. The update server should check whether the code performs any malign tasks before making it available for download.

Secure storage implies that no third party or process running on the server can modify the app code without the consent of the developer. Insecure storage potentially results in malware attacks such as those aforementioned.

Server authentication should be mandatory during the retrieval phase. Mobile devices must be capable of verifying the authenticity of the server hosting the code updates to avoid malicious code being downloaded in the context of the client app.

Our assessment also focuses on how the update strategy deals with *privileges* (or permissions), as the amount of sensitive personal data that can be leaked by an app is highly dependent upon the privileges it is granted. Therefore it is crucial to implement measures that maintain the number of privileges granted under control. This property assesses what permissions are granted to the app if the update plugin is installed.

The overall security of the update strategy also depends on the *quality and codebase of the additional software* that must be installed to support updates. Some update strategies rely solely upon OS software components, while others require the installation of additional software plugins that may be composed by integrating third party software libraries. Integrating them can also increase the number of permissions that are required to run the app. In practice, both plugins and the software libraries on which they rely may contain vulnerabilities. In its turn this can increase the impact of successful attacks both from privacy as well as security point of view.

The *storage location* of the mobile device may also have an impact on the security status of the update strategy. Code can either be stored in shared memory or solely be accessible by the app itself, which obviously provides different levels of security.

 ISBN: 978-1-61208-568-5

B. Quality Criteria

The set of quality criteria is depicted in Figure 2. They primarily focus on user experience, performance, and overall code quality.

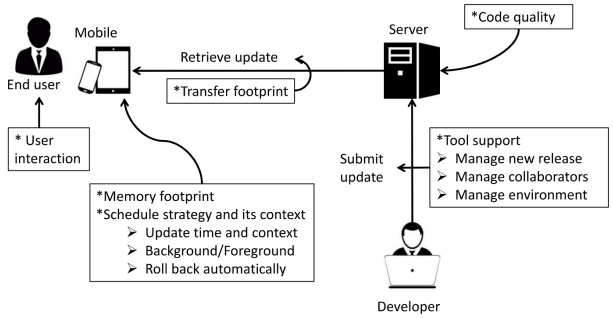

Figure 2: Quality Parameters

For each update strategy, the extent to which *user interaction* is required is evaluated. Either the update can be performed fully transparently, or consent must be explicitly given by the user. If user interaction is required, the type and quality of information returned to the end user is evaluated. Similar, the properties of the *schedule strategy and its context* are evaluated. Such properties can influence user experience. First, the update time and context may differ between alternative approaches. For instance, large (automatic) updates over an xG network are associated with substantial costs and consume unnecessary bandwidth, especially in a roaming context. Second, the update process can either lock the application until finished, or allow the user to use the application during the update. Finally, the possibility of rolling back unstable updates is investigated. This may be required if an update results in crashes on certain devices.

Memory and transfer footprint define the amount of storage required to store both the plugin and updated code and the bandwidth needed to retrieve the update, respectively. Compressed code can be transferred more efficiently and hence has less impact on the time required to install the update than non-compressed code. Similarly, downloading either a complete new version or an increment with respect to the previous version may be needed.

Code quality is often impacted by developer practices and may potentially result in performance decreases or even app crashes after installation. We define code quality in this work from the user's point of view, and evaluate if the mechanism's characteristics meet the user's needs [24]. Code analysis performed by update servers before the update is actually made available and helps avoiding (un)intentionally introduced bugs.

The final quality criterion focuses on the *tool support* available for the update process, like support associated with the management of new releases, enabling collaboration, or tools for covering the entire development cycle (such as design, implementation, and testing).

V. Results and Comparison

This section details the results of the assessment of the update strategies introduced in Section III. First, a general overview of each strategy is provided, after which the security and quality criteria are discussed. In the remainder of this work, the update strategies are denoted as shown in Table I.

Table I: UPDATE STRATEGY NOTATIONS

Update Strategy	Notation
Installation	I
Google Play	I1
Cordova-plugin-app-update	I2
Storage	S
Cordova-plugin-code-push	S1
Cordova-hot-code-push-plugin	S2
Cordova-plugin-meteor-webapp	S3
Cordova-plugin-dynamic-update	S4
Loading	L

A. Overview of update strategies

1) Installation (I): New application code can be stored in the commercial store managed by the OS provider or in another marketplace selected by the developer. Software on the mobile can poll whether a new version is available or new version notifications can be sent to the mobile. Within this class, an entire new version is typically submitted by developers and can subsequently be downloaded.

Developers submit new versions of an app to the standard *Google Play* (I1) store after having assigned it a higher version number. Google reviews the application code using two different approaches. First, an automatic malware scan is performed. Second, a human reviewer investigates possible violations with Google policy rules [25]. Two strategies can be adopted to retrieve a software update: apps are either updated automatically by the Google Play application or the update process is manually controlled by the owner of the device. Note that user consent is always asked when new permissions are required. To save bandwidth, only files differing from the previous version installed on the device are downloaded [26]. Finally, the *cordova-plugin-app-update* (I2) supports the installation of new executables without visiting the default app store, even though the OS does not support this behavior by default and even discourages it. For this, device owners must modify OS configuration settings and the execution of installation files from unknown sources must be allowed. The content of the file is not reviewed, unlike the Google Play strategy. When the app initializes, the plugin checks the version number of the installed app against that of the version available at the update server. A new APK is installed if an update is available.

2) Storage (S): Native mobile apps are typically installed in a read-only folder, which means that only the OS may alter its content. Cordova uses this folder only to store the core files of mobile apps (the native code), including the Webview. This Webview can be pointed to a folder with write permissions and the Web code is then stored and updated via this folder. Any updates to core files can only be performed by submitting and installing a new version. Each plugin follows a slightly different strategy, as discussed next.

The *cordova-plugin-code-push* (S1) checks for updates at a dedicated server and returns a URL from which the latest update can be downloaded. The user must give their consent after which a ZIP file containing the new web code can be downloaded.

 ISBN: 978-1-61208-568-5

The *cordova-hot-code-push-plugin* (S2) checks for updates by reading a configuration file from the server containing a hash of all web code files. The plugin identifies the files that were changed and only updates those files. When the application is launched for the first time, all web code files are copied in the writable *www* folder. The *cordova-plugin-meteor-webapp* (S3) employs a similar approach.

Finally, the *cordova-plugin-dynamic-update* (S4) downloads the entire content of the *www* folder each time an update is available. The previous update of the application is overwritten.

3) Loading (L): App updates occur almost instantly. The app is launched using a browser that loads the content from the server and the only requirement is for the developer to submit the new code to the server. The primary disadvantage of this strategy is that the application only works when an Internet connection is available. In the worst case scenario, the mobile device must contact the server every time the user requests a new page.

B. Security parameter assessment

Malicious individuals may exploit vulnerabilities and thereby compromise the security of mobile app updates. Bad design decisions as well as weak implementations can result in security vulnerabilities. Table II summarizes the results of the analysis of the security concerns listed in the previous when assessing update strategies.

Table II: EVALUATION OF STRATEGIES ACCORDING TO SECURITY PARAMETERS

Criteria/Strategy	I1	I2	S1	S2	S3	S4	L
Submit update							
Accountability	yes	no	no	no	no	no	no
Secure transport (submit)	yes	no	yes	yes	no	no	no
Developer Authentication	yes	no	yes	yes	no	no	no
Access control	yes	no	yes	yes	no	no	no
Secure storage (server)	yes	no	yes	yes	no	no	no
Code analysis (server)	yes	no	no	no	no	no	no
Retrieve update							
Secure transport (retrieve)							
HTTPS forced	yes	no	yes	yes	no	no	no
SSL Certificates checked	yes	yes	no	yes	yes	yes	yes
Hash check after download	yes	no	yes	yes	no	no	-
Server Authentication	yes	no	no	no	no	no	no
Privileges	no	yes	yes	yes	yes	yes	no
Code verification (device)	yes	no	no	no	no	no	-
Additional software	no	yes	yes	no	no	no	-
Secure storage (device)	yes	yes	yes	yes	yes	yes	-

Two approaches are possible for setting up an update server. First, the update server can be fixed in the update plugin. This means that each developer that uses the plugin must push updates to that third-party server. The server often represents a reliable Cloud platform (examples of such include Android Play Store or Microsoft Azure) associated with a series of advantages like high availability and small security vulnerability due to regular software updates. Second, some plugins enable developers to select their own storage server. Therefore, security is highly dependent upon both the reputation of the developer and selected server. The plugins that fix a specific update server (I1, S1, and S2) also provide a secure communication channel between the developer and the server. Other strategies are dependent upon the server hosting code updates. Storage security also relies on the trustworthiness of the server that hosts the updates. Similarly, some plugins rely on existing back-end infrastructure and, hence, benefit from the access control procedures offered by the used infrastructure. Only the owner (and possibly collaborators) can modify code within a particular application. The quality of other plugins depends on the particular update server selected. Therefore, if the developer is free to choose an update server, they must select one that has good security practices. Note that the end user is unaware of the selected server.

For the security of the end user, it is also important that the update reaches the device securely. The Android Play store is tightly integrated with the underlying operating system and offers all required security measures, as a secure HTTPS connection is strictly used for all communication.

Secure communication is not always guaranteed when using a plugin approach. In fact, some plugins do not force the use of HTTPS on the developer. S1 and S2 are an exception to this. They respectively make use of the Amazon and the Microsoft Azure platform to distribute updates, and therefore always make use of HTTPS. In the other cases, the responsibility to use HTTPS lies with the app developer.

When using an HTTPS connection, it is mandatory to check the SSL certificates before accepting communication. Plugins I2, S2 and S4 have custom native code for downloading the update. They make use of Android's `HTTPUrlConnection`, `UrlConnection` and `HTTPClient`, respectively, and are therefore secure when HTTPS is used. Android provides the necessary checks to secure the connection once an HTTPS domain is detected in the URL (for more information [27], [28]) Plugin S1 contains no native code to download the update but instead automatically includes the *file-transfer-plugin* in the application when installing the update plugin, and contains Web code to call the plugin to download the update. For developers, it is important to know that the *file-transfer-plugin*'s `download()` method contains an "AllowAllHosts" parameter. When this parameter is set to true, it overrides Android's HostnameVerifier's [29] `verify()` method to always return true. In this case, SSL certificates are blindly accepted without any checks, which raises severe security threats. Plugin S1 sets this parameter to true and is therefore not secure. Developers using this plugin should manually change this variable to false in the its web code in order to ensure secure communication. Plugin S3 contains no code for the actual download of the update. The straight-forward way to download the update in this case is to include the *file-transfer-plugin* in the application manually. Some example code provided by the developers of the plugin also applies this strategy. In this example, no AllowAllHosts parameter is specified, thus the default value is false. Hence, developers can use this example to provide secure communication.

After downloading the update, I1, S1 and S2 make use of a hashing function to determine if nothing was changed during the download. This acts as an additional measure against MiTM attacks, and also ensures that no communication errors occurred.

When loading web content in the Webview or a browser (L), the security depends only on whether HTTPS is used or not. When HTTPS is used, the communication is handled by the Webview and is therefore secure.

 ISBN: 978-1-61208-568-5

Strategy I1 supports developer accountability. It relies on the OS platform builder and requires one to update the submission via the Android store, and developers must sign the code before submitting it to the server. This procedure verifies if the files were submitted by a particular developer (or organization). In addition, Play store examines the code submitted by the developer before publishing the application on its store. In practice, it looks for malicious code and violation of system rules in the application files. If an unacceptable security or privacy threat is discovered the application is not published, the developer can be judged and their profile removed from the store. Other storage platforms do not explicitly mention code revision. Except for the Google Play Store downloads and updates [30], no other strategy applies client side code verification, despite there being some tools and methods currently available to execute it. Such tools could mitigate threats introduced by selecting unreliable update servers.

The Google play store (I1) is the only update mechanism that requires the user to authenticate to the update server. Android requires the user to be logged in before allowing downloading or updating applications.

The OS coordinates code updates within strategy I1. It saves new files in a folder that cannot be overwritten (unless a new code update is available) by the application or possible malware. All other strategies retrieve updates stored in memory only accessible by the app itself. This means that other, potentially malicious apps, cannot access or modify that code. Hence, once updates are installed on the mobile device, the security depends upon the trustworthiness of the OS version.

Update plugins often require extra privileges to support their tasks. Two permissions that every Cordova plugin requires are `internet` and `write_external_storage` permission. The former allows apps to open network sockets and is enabled by default in the Cordova framework; the latter allows apps to write content to external storage on the device. These permissions can be abused by locally installed untrusted code which downloads malicious content from the Internet before writing it to the device and altering the application to make it use this malicious content. The `mount_unmount_filesystems` permission is used by I2 to manage the file systems for removable storage. S2 also requires the `access_network_state` and `access_wifi_state` permissions for accessing information concerning network status.

Additional Cordova plugins (also called dependencies) are installed automatically when strategies I2 or S1 are applied. I2 installs an additional Cordova plugin for accessing the OS version. S1 relies on a set of seven other plugins that handle sending notifications to users, accessing configuration information from the device, handle runtime permissions, file management, among others. Note that installing such plugins weakens the privacy properties of the app and weak implementations may ultimately lead to vulnerabilities that can, in turn, compromise app security. Although S3 does not automatically introduce any dependencies, it requires additional plugins to work properly.

C. Quality parameters assessment

Table III provides an overview of the quality related parameters for each update strategy. Strategies S3, S4, and L neither require user confirmation nor notification of the end user regarding code updates. All other strategies ask the user to confirm the update. I1 also provides feedback to the user; I2, S1, and S2 do not inform the user about the scope of the updates, although certain code updates may negatively impact the user's privacy or security (as learned from the security assessment).

Table III: EVALUATION OF STRATEGIES ACCORDING TO QUALITY PARAMETERS

Criteria/Strategy	I1	I2	S1	S2	S3	S4	L
User interaction	yes	yes	yes	yes	no	no	no
Schedule strategy and its context							
Update time and context	req.	any	any	any	any	any	any
Lock the application	yes	yes	no	no	no	no	-
Roll back automatically	yes	no	yes	yes	yes	no	yes
Memory and Transfer footprint	++	++	+	-	-	+	++
Code Quality Verification	yes	no	no	no	no	no	no
Tool Support	yes	no	yes	yes	no	no	no
Manage new release	yes	-	yes	yes	-	-	-
Manage collaborators	yes	-	yes	no	-	-	-
Manage environment	yes	-	yes	yes	-	-	-

I1 supports automatic updates of non-active applications. Other approaches typically check whether an update is available when it launches or resumes, automatically. Alternatively, the developers may provide a button in the app to begin the update process. Strategy L receives updates when the page is (re)loaded. Cordova update strategies S_x do not enforce any constraints on running applications. In contrast, I1 and I2 lock the application until the update process has ended.

Strategy I2 and S4 do not support roll backs to a previous valid version in case of crashes caused by the update. When such a situation arises the end users must reinstall the application. All other strategies perform roll backs automatically when an error occurs. Crashes in strategy I1 occur rarely, due to heavy code revision by Google and the reporting of unexpected behavior by the end-users [25]. This strategy guarantees that a stable version of the application is available on the store until the developer provides a valid new version.

A large amount of code and data is potentially downloaded during the update process. Strategies I1 and S2 support contextual constraints related to updates. For instance, these strategies permit developers to specify that updates may only occur over WiFi. Other strategies do not support this control and, hence, the update can be downloaded via the mobile network operator, thus potentially incurring additional costs.

Most strategies do not ensure code quality thresholds. Indeed, only strategy I1 performs code quality analysis to ensure stable app behaviour. Thus, other strategies may lead to poorly implemented features, wasted processing time and memory. Such situations can result in bad user experiences and the misuse of the application given the resource constraints of the device.

Strategies I1, S1, and S2 offer tool support to aid developers when building a new application release. All these tools provide a feature to manage releases and make it easy to uploade new releases to the server, and make them available for download. The developer must change the version number of the application in strategy I1, which also allows automatic code signing before upload. In contrast, the tool used by strategies S1 and S2 changes the version number automatically during the upload process. These tools also calculate the hash

of the files in the current version, enabling mobile devices to download only the altered files, and hence decreasing bandwidth. I1 and S1 also support collaboration within a team of developers via *developer profiles*. The application owner is responsible for adding and removing collaborators and may also identify who submits particular code updates. Strategy S2 does not offer this feature and only the owner may upload releases on the server. All tools provide a production and test environment. The latter enables the developer to simulate real world situations.

VI. Evaluation and Reflection

Especially in the case of Cordova applications, man-in-the-middle attacks on code transfers impose severe risks for the security of the end user. Attackers can, for example, insert malicious code in JavaScript files while in transit. The impact of such a MiTM attack strongly depends on the capabilities and privileges of the application. For example, many Cordova applications include several plugins to provide desired functionality such as access to the GPS, contacts, pictures and the camera. Malicious code, injected by attackers, also has access to the API's of these plugins, and can use them to steal personal information. In order to demonstrate the mechanisms and the risks of a MiTM attack, the next paragraph describes an example setup of such attack. It demonstrates a straightforward, realistic setup for executing MiTM attacks on regular HTTP connections.

The setup is displayed in Figure 3 and consists of two workstations, connected to the same network, and a mobile device.

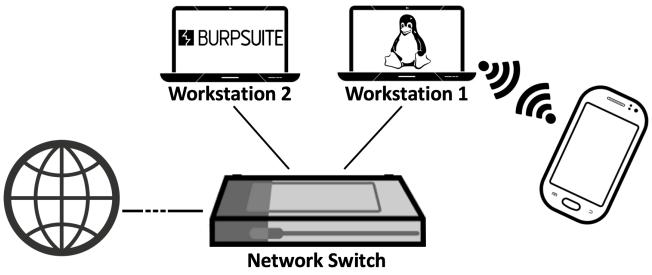

Figure 3: MiTM demonstration setup.

- **Workstation 1** contains a classic Linux distribution. This workstation is set up to create its own wireless network, thereby acting as a public wireless access point. All incoming wireless regular communication (wlan0) is redirected to workstation 2.
- **Workstation 2** runs a proxy tool. BurpSuite [31] was used in our setup. It listens for the traffic forwarded by workstation 1. An attacker can now read and alter all requests and responses.
- The **mobile device** is connected to the Internet via the wireless network provided by Workstation 1 and contains the application under attack. All HTTP traffic on the mobile device is now routed transparently through both workstations.

This attack describes the situation when a user accesses a public access point. Traffic can be eavesdropped and altered without the user noticing. For the sake of clarity, workstations 1 and 2 are displayed as two separate entities. Note that the functionality of both workstations can be combined into a single one. MiTM attacks on unsecure SSL connections (e.g. certificate checks are lacking) work similarly to the demonstrated setup, but require an SSL stripping step in the proxy (Workstation 2).

Our assessment mainly focused on the Android operating system. Nevertheless, most ideas presented in the paper also apply for iOS. In iOS, it usually takes several days before an application is approved and available the app store, hereby making hot code updates even more valuable than in Android. On the other hand, iOS developers have strict limitations for what a hot code update can change in an application [32]. The features and functionality have to be consistent with the intended and advertised purpose of the application as submitted to the App Store. Developers who do not comply with these rules can lose their access to the App Store.Not all Cordova plugins described in this work are available for iOS. Naturally, *Cordova-plugin-app-update* (I2) provides no iOS implementation because it relies on downloading and installing an APK file. Also, *cordova-plugin-dynamic-update* (I4) does not provide an iOS implementation. However, the security risks discussed also apply for iOS. For example, if developers do not secure the communication with the update server, the attack described above can also be executed on iOS applications.

Independently of the targeted operating system, developers have to be aware that plugins might contain unsafe behavior and ignore the best development practices for the platform. Hence, the source code should always be checked before including it in applications. In Android, examples of this are introducing too many permissions and overriding the security mechanisms of the platform, as described in Section V. iOS implementations of plugins can contain similar insecure behavior. For example, the *cordova-plugin-meteor-webapp* plugin disables Apple App Transport Security, introduced in iOS 9 [33].

VII. Conclusion

This paper presented an analysis of hot code updates in Cordova, and compared them against traditional updates through the app store and the loading strategy. Two sets of parameters were discussed in this work. The first group of parameters focused on the security aspect of the updating mechanisms. One of the major concerns when using hot code updates is the secure storage of the update and transition of the update to the device. Some plugins are developed for one specific type of server (e.g. Amazon, Microsoft Azure). In this case, the developer can rely on these mature platforms for securely hosting the update (secure storage on the server and only use HTTPS for communication), and has access to additional tools for managing the update. Other plugins do not offer this complete solution. Hence, the developer is responsible for implementing a secure server to host the update. Besides the server, the plugin is also responsible for secure communication. Developers should check the implementation before installing it in an application. Plugin developers should rely on the underlying platform's APIs as much as possible in order to ensure secure communication.

The second group of parameters focused on the functionality and the usability. The conclusion here is that, in many cases,

 ISBN: 978-1-61208-568-5

hot code updates offer a viable solution for minor updates and bug fixes to an application. Based on the needs of the developer and the application, different plugins can be selected. Some plugins offer a simple but limited API, and are therefore quick and easy to integrate in applications. The disadvantage is that these plugins do not offer support for user interaction such as asking permission from the user and giving the user a visual overview of the progress. Other plugins offer a more advanced API to the developer and allow more customization. Hence, these plugins are also more complex to implement for the developer.

REFERENCES

[1] T. Yan, D. Chu, D. Ganesan, A. Kansal, and J. Liu, "Fast app launching for mobile devices using predictive user context," in Proceedings of the 10th international conference on Mobile systems, applications, and services. ACM, 2012, pp. 113–126.

[2] AppBrain. Number of android applications. [Online]. Available: http://www.appbrain.com/stats/number-of-android-apps [retrieved: October, 2016]

[3] C. Reese Bomhold, "Educational use of smart phone technology: A survey of mobile phone application use by undergraduate university students," Program, vol. 47, no. 4, 2013, pp. 424–436.

[4] T. Petsas, A. Papadogiannakis, M. Polychronakis, E. P. Markatos, and T. Karagiannis, "Rise of the planet of the apps: A systematic study of the mobile app ecosystem," in Proceedings of the 2013 conference on Internet measurement conference. ACM, 2013, pp. 277–290.

[5] S. Amatya and A. Kurti, "Cross-platform mobile development: challenges and opportunities," in ICT Innovations 2013. Springer, 2014, pp. 219–229.

[6] H. Heitkötter, S. Hanschke, and T. A. Majchrzak, Evaluating Cross-Platform Development Approaches for Mobile Applications. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 120 – 138.

[7] P. R. de Andrade, A. B. Albuquerque, O. F. Frota, R. V. Silveira, and F. A. da Silva, "Cross platform app: a comparative study," arXiv preprint arXiv:1503.03511, 2015.

[8] B. R. Mahesh, M. B. Kumar, R. Manoharan, M. Somasundaram, and S. Karthikeyan, "Portability of mobile applications using phonegap: A case study," in Software Engineering and Mobile Application Modelling and Development (ICSEMA 2012), International Conference on. IET, 2012, pp. 1–6.

[9] K. Leuven. Crossmos. [Online]. Available: https://www.msec.be/crossmos/ [retrieved: June, 2017]

[10] M. Latif, Y. Lakhrissi, E. H. Nfaoui, and N. Es-Sbai, "Cross platform approach for mobile application development: A survey," in 2016 International Conference on Information Technology for Organizations Development (IT4OD). IEEE, 2016, pp. 1–5.

[11] C. Rieger and T. A. Majchrzak, "Weighted evaluation framework for cross-platform app development approaches," in EuroSymposium on Systems Analysis and Design. Springer, 2016, pp. 18–39.

[12] M. Willocx, J. Vossaert, and V. Naessens, "Comparing performance parameters of mobile app development strategies," in Proceedings of the International Workshop on Mobile Software Engineering and Systems. ACM, 2016, pp. 38–47.

[13] A. Pazirandeh and E. Vorobyeva, "Evaluation of cross-platform tools for mobile development," 2015.

[14] G. S. Kearns, "Countering mobile device threats: A mobile device security model," Journal of Forensic & Investigative Accounting, vol. 8, no. 1, 2016.

[15] P. De Ryck, L. Desmet, F. Piessens, and M. Johns, Primer on client-side web security. Springer, 2014.

[16] S. Vashisht, S. Gupta, D. Singh, and A. Mudgal, "Emerging threats in mobile communication system," in Innovation and Challenges in Cyber Security (ICICCS-INBUSH), 2016 International Conference on. IEEE, 2016, pp. 41–44.

[17] M. La Polla, F. Martinelli, and D. Sgandurra, "A survey on security for mobile devices," IEEE communications surveys & tutorials, vol. 15, no. 1, 2013, pp. 446–471.

[18] F. Peijnenburg, "Security in android apps," 2013.

[19] A. Bhardwaj, K. Pandey, and R. Chopra, "Android and ios security-an analysis and comparison report," Int'l J. Info. Sec. & Cybercrime, vol. 5, 2016, p. 30.

[20] A. Cordova. Plugin search - apache cordova. [Online]. Available: http://cordova.apache.org/plugins/ [retrieved: June, 2017]

[21] Google. Google play. [Online]. Available: https://play.google.com [retrieved: June, 2017]

[22] Microsoft. Codepush. [Online]. Available: http://microsoft.github.io/code-push/ [retrieved: June, 2017]

[23] Github. Github - nordnet/cordova-hot-code-push-cli. [Online]. Available: https://github.com/nordnet/cordova-hot-code-push-cli [retrieved: July, 2016]

[24] B. Kitchenham and S. L. Pfleeger, "Software quality: the elusive target [special issues section]," IEEE software, vol. 13, no. 1, 1996, pp. 12–21.

[25] S. Perez. App submissions on google play now reviewed by staff, will include age-based ratings. TechCrunch. [retrieved: March, 2015]

[26] A. Morris. Improvements for smaller app downloads on google play. Android Developers Blog. [retrieved: July, 2016]

[27] Google. Httpurlconnection - android developers. [Online]. Available: https://developer.android.com/reference/java/net/HttpURLConnection.html [retrieved: June, 2017]

[28] ——. Security with https and ssl - android developers. [Online]. Available: https://developer.android.com/training/articles/security-ssl.html#HttpsExample [retrieved: June, 2017]

[29] ——. Hostnameverifier - android developers. [Online]. Available: https://developer.android.com/reference/javax/net/ssl/HostnameVerifier.html [retrieved: June, 2017]

[30] T. N. Web. Google describes how android 4.2's app verification checks your downloads for malware. [retrieved: November, 2012]

[31] PortSwigger. Download burp suite - portswigger. [Online]. Available: https://portswigger.net/burp/download.html [retrieved: June, 2017]

[32] Apple. Apple developer program information. [Online]. Available: https://developer.apple.com/programs/information/Apple_Developer_Program_Information_8_12_15.pdf [retrieved: December, 2015]

[33] ——. ios 9.0. [Online]. Available: https://developer.apple.com/library/content/releasenotes/General/WhatsNewIniOS/Articles/iOS9.html [retrieved: June, 2017]

 ISBN: 978-1-61208-568-5