Reducing Attack Surface on Cordova-Based Hybrid Mobile Apps

Mohamed Shehab   Abeer AlJarrah

Department of Software and Information Systems
University of North Carolina at Charlotte
Charlotte, NC, USA
{mshehab,aaljarra}@uncc.edu

Abstract

Hybrid mobile application development is increasingly being adopted by the mobile development community since it provides the answer to the challenge of having the right mix of accessibility to mobile native features at an affordable development cost. Apache Cordova library is an example of a middle-ware that enables developers of different mobile operating systems to access mobile native features through web frameworks, such as HTML and JavaScript, which at the same time introduces several security challenges. In this paper, we highlight current security setting limitations of hybrid mobile frameworks and propose a policy based approach to provide limited access to the different pages/states of the app to mitigate the effect of possible attacks. In addition, we downloaded and analyzed 622 real hybrid apps, and presented settings and security statistics.

Categories and Subject Descriptors D.4.6 [*Security and Protection*]: Access controls; D.2.9 [*Software Management*]: Software development

Keywords Cross-Platform, Apache Cordova, Access Control, Hybrid, PhoneGap

1. Introduction

The revolution of mobile technologies in terms of hardware and software has created a shift towards investing more in this landscape. One of the latest revolutions in this area was the introduction of cross-platform mobile hybrid application development which enables developers to code their apps using the standard web stack (HTML, JS, and CSS) that can access native device functions like-native then packages these apps towards multiple mobile platforms. This approach proved to be a promising development solution for building generic apps, supported by ready non-commercial tools[10, 13]. As this approach is expected to be the future of mobile development [7], many platforms are competing in the cross-platform hybrid mobile development market based on the services the platform provide to the developer such as supported platforms, access to native device functions, backend support and other performance issues.

Apache Cordova is a middle-ware library that enables access to native device functions by providing a set of APIs that supports interaction with native device functions or "plugins" through Javascript. This library is a key component of many hybrid platforms including PhoneGap, IBM Worklight, and App Builder.

In this paper, we analyze the architectural and security details of the Apache Cordova based frameworks and highlight possible threats on Apps generated using these frameworks. This study is based on current implementation of the latest Apache Cordova version (3.5.0) released on (June 12, 2014). Security of such platforms has recently gained the attention, several studies [1, 4, 8] have analyzed possible security threats and have proposed possible solutions. In this paper, our study focuses on the plugin access model implemented by Cordova framework, which to the best of our knowledge has not been addressed nor analyzed in previous studies. It is important to analyze what security models are implemented to protect user's device against possible compromises, especially since all the threats of web based apps can easily be transformed to hybrid apps. The current security model of the Cordova framework provides a coarse-grained based plugin access control model where access is granted at the app level. In this paper we propose a page based access control model that enables developers to limit the access granted to the different pages of the app. We summarize our contributions as follows:

- We propose a framework that enables developers to have more control over the plugins exposure to each page of the app, and to specify access policy rules to control plugin access.

MobileDeLi '14, October 21, 2014, Portland, OR, USA.
Copyright © 2014 ACM 978-1-4503-2190-7/14/10...$15.00.
http://dx.doi.org/10.1145/2688412.2688417

- We analyze a repository of 662 hybrid (Cordova-based) Android Apps in order to understand developers practices, apps structure and plugin usage patterns.

The rest of the paper is organized as follows: In Section 2, we provide a brief background of hybrid mobile frameworks and focus on the details related to the Apache Cordova library. Section 3 discusses the different security threats in Cordova-based hybrid mobile frameworks. Section 4 details our proposed page based access framework. Section 5 presents detailed analysis of the downloaded mobile hybrid apps. Section 6 provides discussion of the related work. Section 7 discusses the future extensions and paper conclusion.

2. Preliminaries

Cross-platform hybrid mobile framework manages the connection between the native app and the embedded web browser component. It enables the app's JavaScript to communicate with the native application programming interfaces (APIs) to access native resources, such as the network, camera, GPS and contacts. In addition, several frameworks provide several settings to control and setup the communication channel between the embedded web browser component and the hosting native app.

The web browser component is a user interface component that can be embedded in a native mobile app to render (HTML/CSS) content and execute JavaScript. This component is available in different mobile frameworks, WebView in Android, UIWebView in iOS, and WebBrowser in Windows Phone. In this paper we will focus on Android platform because of its openness; however, our discussion is applicable to other platforms as well. The WebView component uses the WebKit rendering engine to display web pages, in addition the WebView component executes the scripts (JavaScript) imported or included in the page. The WebView is embedded in a native app that can control the embedded WebView. For example, the native app can load a URL in the WebView or execute JavaScript in the currently rendered page. Figure 1 demonstrates how the WebView `loadUrl` method can be used to load a specific URL and execute JavaScript in the context of the currently rendered page.

```
WebView webview = new WebView(this);
setContentView(webview);
webview.getSettings().setJavaScriptEnabled(true);
webview.loadUrl("http://www.uncc.edu");
webview.loadUrl("javascript:alert('hello');");
```

Figure 1: WebView example

The developer can customize the WebView's behavior through WebView clients (WebViewClient and WebChromeClient), which can be used to register event handlers to respond to WebView events such as `onPageStarted`, `onPageFinished` and `onJsAlert`. In addition, the native app is able to receive data directly from the embedded WebView by injecting a native Java object into the WebView. The object is injected into the currently loaded JavaScript context and the object is accessible through the supplied name. Through this Java to JavaScript interface, the injected Java object's methods are accessible from JavaScript. Figure 2 demonstrates how a native object is injected into the WebView using the `addJavascriptInterface` method.

```
class JsObject {                                                    Native Java
     public String save(String data) {
          //save data
          return "value";
     }
}
webview.addJavascriptInterface(new JsObject(), "injectedObject");
```

```
                                                                    JavaScript
var value = injectedObject.save("data");
```

Figure 2: WebView javascript interface

2.1 Apache Cordova Library

There are several cross-platform frameworks that implement Apache Cordova library, we focus on the PhoneGap framework which is a popular framework that uses the Apache Cordova engine as a middleware that provides APIs to establish communication channels between native code and JavaScript. In this section, we discuss the details of the hybrid PhoneGap apps that target the Android framework, and we explain the details of the Cordova implementation in the Android framework. Similar overall context applies for other mobile frameworks that use the Cordova library. The Cordova library defines the native (Java) to JavaScript interfaces through the WebView interface. The native library provides native APIs that are developed as native classes which are referred as *plugins* or *features*. These plugins include the native code required to access native device resources such as location services, camera, and contacts. In addition to the PhoneGap provided plugins developers can create and include customized third party plugins, which requires the definition of both native plugin libraries and their corresponding JavaScript interfaces. In PhoneGap the app configuration file (`config.xml`) is used to specify the app settings, such as the plugins to be included, the application orientation (landscape, or portrait) and many other settings. Plugins are included in the app by declaring `feature` tags specifying the plugin library in the `config.xml` file, figure 3 shows an example config file which includes the Accelerometer and Compass plugins. Note that the `org.apache.cordova.devicemotion.AccelListener`, is the native plugin class name which contains the plugin methods that access the native accelerometer APIs.

When the app starts the main activity, which hosts the embedded WebView, initializes several components, loads the app configuration, and loads the apps startup HTML page. The `CordovaActivity` (also known as `DroidGap`) class is the main app entry point. The following are the native and client

```xml
<widget id="com.phonegap.helloworld" version="1.0.0">
    <name>Hello Cordova</name>
    <description>A sample Apache Cordova app</description>
    <access origin="*" />
    <content src="index.html" />
    <author email="aaljarra@uncc.edu" href="http://liisp.uncc.edu">
        LIISP Team
    </author>
    <feature name="App">
        <param name="android-package" value="org.apache.cordova.App" />
    </feature>
    <feature name="Accelerometer">
        <param name="android-package"
          value="org.apache.cordova.devicemotion.AccelListener" />
    </feature>
    <feature name="Compass">
        <param name="android-package"
          value="org.apache.cordova.deviceorientation.CompassListener" />
    </feature>
    <preference name="loglevel" value="DEBUG" />
</widget>
```

Figure 3: PhoneGap config.xml file

(JavaScript) components included in a PhoneGap project that are needed to enable the communication between native side and web side:

- `ExposedJsApi` (Native): Global native object shared with the WebView's JavaScript through JavaScriptInterface.
- `CordovaWebview` (Native): The customized WebView. During initialization the native `ExposedJsApi` is registered in the WebView by using `addJavascriptInterface`.
- PhoneGap Client Library (Client): The JavaScript library that contains the PhoneGap client functions, and that decodes the client API calls to javascript messages to be sent to the `CordovaWebView` instance.
- `CordovaChromeClient` (Native): A WebView client that is attached to the `CordovaWebview` which is used to register event handlers associated with the WebView.
- `PluginManager` (Native): The central component in app's operation, it is responsible for the initialization of the included app plugins, manages the mapping of the client API calls to the corresponding native plugin APIs.

The client app (JavaScript) is able to send and receive messages from the native components through the established PhoneGap interfaces. When the client app issues a request to listen to the compass heading, this is sent as a message to the native code to execute the corresponding plugin methods and ultimately send the compass heading data to the client app.

Figure 4 illustrates the execution flow from the client code to the native code and then back to the client code. The flow starts when the client (JavaScript) app requests to access a native resource (Step 1), in this example the app is requesting to listen to the compass heading by calling the client method `compass.watchHeading()` and providing the method with parameters which include a callback function for success, a callback function for error and an array of options (if needed). The `compass.watchHeading()` is a wrapper method to an `exec()` function that triggers

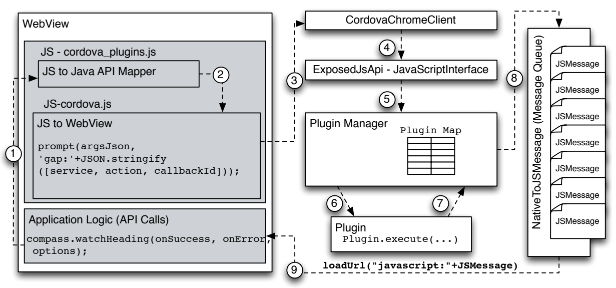


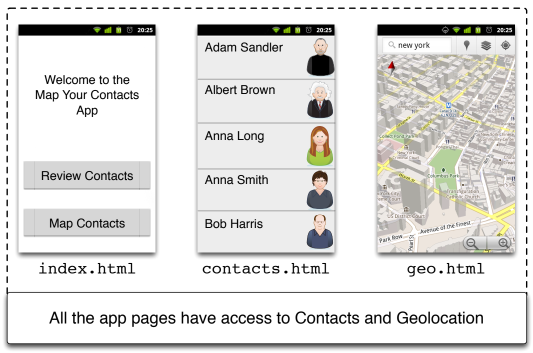
Figure 4: Plugin access control execution flow

a JavaScript prompt event (Step 2) that is captured by the `CordovaChromeClient` which is initialized to handle to the prompt event, see Step 3. In previous PhoneGap versions (beforee 2.3), the `exec()` function used the injected native object `exposedJsApi` but in later versions the JavaScript prompt event was used. The event handler in the `CordovaChromeClient` calls the `ExposedJSApi.exec()` method and passes it all the received parameters indicating the requested service, action, callback ID and other arguments (Step 4). The `PluginManger.exec()` method is called to resolve and call the plugin that should be activated (Step 5). During the app initialization the `PluginManager` loads the `config.xml` and creates a plugin mapping table that maps the service to a native plugin class. As indicated in Figure 3, the *feature* tags are used to specify the native packages for the included plugins. The `PluginManager` locates and instantiates the corresponding plugin class and the indicated action (method) is executed by the `PluginManager` (Step 6). The plugin result is returned to the `PluginManager` (Step 7), which then enqueues the returned result as a JavaScript encoded message into the message queue (Step 8). The message is dequeued and loaded in the WebView which executes the corresponding callback function with parameters being the result of executing the requested plugin (Step 9).

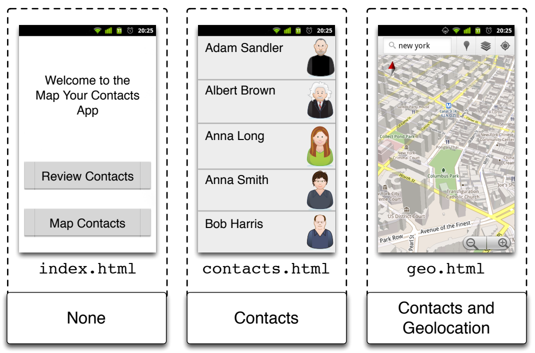
It is important to note that plugins vary in their sensitivity and access to private information. Some plugins require no native permissions and others require several permissions in order to execute. The PhoneGap has an active community and most enhancements are moving towards more security checks and requiring more control on plugin access. After releasing Cordova 2.8.0 (06/12/2013), the PhoneGap documentation added a privacy guide, which contains recommended policies to be used by developers when using plugins specially those accessing private information. Despite of all the controls provided by PhoneGap to manage plugin access, it is still possible to compromise apps and bypass these checks as will be discussed in the following section.

3. Possible Security Threats

To configure the required permissions for a hybrid app there are two main stages. The first step is to set up the permissions granted to the native application hosting the hybrid



(a) Current Cordova plugin access Model



(b) Proposed Least Privilege Approach

Figure 5: Example Multi-Page App and Access Models

app, in Android for example, the native application permissions should be declared in the `AndroidManifest.xml` file. The second step is to set up the platform specific configuration file (`config.xml`) to specify the plugins that should be included in the app. Developers using Cordova versions before 3.1.0 are required to configure the project manually and this can easily cause confusion and result in misconfigured and over-privileged applications. In addition, projects created by these versions include all the plugins by default in the `config.xml` file, which eliminates the second step, but results in having plugins declared in the `config.xml` that are not required by the app, hence, increase the attack surface of the app. In addition, some plugins like the accelerometer plugin do not required any permissions to execute which enables attacker to access this plugin easily by calling it through the Cordova bridge.

Later versions of Cordova use a minimalistic approach since by default the `config.xml` file does not include all the plugins by default and the application developer is required to add the required plugins. In addition, Cordova provides a Cordova Command-line Interface (CLI) scripting tool to streamline the management of device level features and to configure the project settings and permissions. In Cordova, plugin access is set on the app level, i.e. declaring a plugin in the `config.xml` file implies that the plugin is accessible to any local and dynamic JavaScript files loaded in the WebView component. Thus malicious scripts are able to execute with the same privilege irrespective to the page in which they are loaded or executed.

The previous threats were due to the current implementation of Cordova security access model, other threats arise due to the WebView implementation. The WebView's `loadUrl()` method can be used to load content and scripts into the WebView. An app can load pages and scripts from local and external sources which introduces several vulnerabilities [9] as remote pages and scripts could be easily loaded and gain access to sensitive information throughout the native device services. In addition, dynamically loaded JavaScript can easily introduce malicious code into the app which will not be detected by the application vetting process. To be able to control the source of loaded content Cordova uses a domain whitelisting security model. The developer should specify the whitelist of allowed domains, which is a list of trusted URLs. The `access` element in the `config.xml` file specifies the allowed domains. The default policy is to allow all local and external domains, as indicated in the configuration file in Figure 3. The wildcard `<access origin="*"/>` allows access to any external resource. In addition, it is possible to allow access of resources from specific domains, for example, `<access origin="https://www.google.com"/>`.

There are many threats [2, 3, 9] associated with using WebViews in mobile apps. The WebView `loadUrl` and `addJavascriptInterface` methods can easily be used to introduce back channels to give access to malicious apps [9]. For example, `loadURL` can also be used to inject malicious javascript code into the WebView. Similarly event hijacking can be performed by registering custom malicious event handlers in the WebView client component, which can enable attackers to override the app expected behavior. To reduce the risk of these attacks, Cordova provides the developer with configurations to white list the source of the loaded pages and scripts and to control the plugins to be included in the app. The app developers should carefully configure their hybrid apps to ensure the correct security configurations.

4. Proposed Least Privilege Approach

The Cordova access policy allows the developer to specify a global policy to be adopted for the whole app, if the app has multiple pages then all the pages have the same access to all the plugins included in the app. This approach does not ensure the principle of least privilege [11], since extra permissions and access to plugins are granted to loaded pages that do not require such accesses. Figure 5(a) shows our running example Map your Friends app which is composed of three pages `index.html`, `contacts.html` and `geo.html`. The `index.html` page displays the application loading screen, it does not require any permissions/ plugin. The `contacts.html` page displays the user's contacts stored on the mobile device which requires permissions to access the contacts. The `geo.html` displays the user's location on a map and allows users to map their friends living near their current location which requires access to the contacts and location services. The current Cordova policy will grant the three pages `index.html`, `contacts.html` and `geo.html` access to the contacts and location services which doesn't obey the principle of least privilege.

We propose a framework that enables developers to build and enforce page-based plugin access policy by slightly modifying the Cordova library. Figure 5(b) shows the proposed approach when applied to the Map your Friends example app, where the `index.html` is granted no plugin access, the `contacts.html` is given access to only the contacts plugin and the `geo.html` page is granted access to the contacts and geolocation plugins.

In order to implement the proposed framework in the context of Cordova-based app, we require the app undergo two main stages, namely the build and enforce stages. The "build" stage, is a monitoring stage in which the application's access policy is composed and this stage is performed during the application development and testing. The "enforce" stage, is the policy enforcement stage in which the composed policy in the build stage is enforced, this stage is activated when the app is deployed. We extended the `config.xml` file to include a new `<policy />` element in order to identify the current application stage which can be build or enforce, see Figure 6.

```
<widget id="com.phonegap.helloworld" version="1.0.0">
    <name>Hello Cordova</name>
    <description>A sample Apache Cordova app</description>
    <access origin="*" />
    <content src="index.html" />
    <author email="aaljarra@uncc.edu" href="http://liisp.uncc.edu">
        PhoneGap Team
    </author>
    <feature name="App">
      <param name="android-package" value="org.apache.cordova.App" />
    </feature>
    <feature name="Contacts">
      <param name="android-package"
          value="org.apache.cordova.contacts" />
    </feature>
    <feature name="Geolocation">
      <param name="android-package"
          value="org.apache.cordova.geolocation.GeoBroker" />
    </feature>
    <policy stage= "build" />
</widget>
```

Figure 6: Policy stage in config file

To implement the proposed approach, the following challenges should be addressed. First, how to build the plugin access policy? Second, where to store the proposed policy rules? Third, where to implement the required reference monitor? To address the first challenge, our proposed approach is based on monitoring the app's behavior during the "build" stage and to record the plugin API call traces for each page, which includes the page URL, plugin name, and plugin action. The "build" stage is performed during the development and the testing phase and all accesses performed during these phases are considered valid plugin rules, as a result we are able to automatically build the policy rules without requiring the developer to manually enter the policy rules, which would avoid confusion and error. The main assumption is that through app development and testing most if not all possible plugin calls are covered as a normal testing procedure. Figure 7(a) shows the build stage details. To address the second challenge, the monitored app behavior will be recorded, hashed and stored in a local in-app SQLite database. The rule hashing is computed by computing the secure hash (MD5 or SHA1) of the string concatenation of the page url, plugin name, and the plugin API, the hash is stored in the SQLite database. The access control model follows the closed world assumption, where if access is not explicitly specified then it is assumed it is not granted. Once the app development and testing is completed the app is released with the database having the access policies, the policy flag is set to "enforce" which stops writing to the database and enables the authentication of every plugin access requests against the policies saved in the SQLite database, Figure 7(b) describes the enforcement stage. The change is transparent to the developer, since it doesn't require the developer to perform any extra steps regarding setting or enforcing policies.

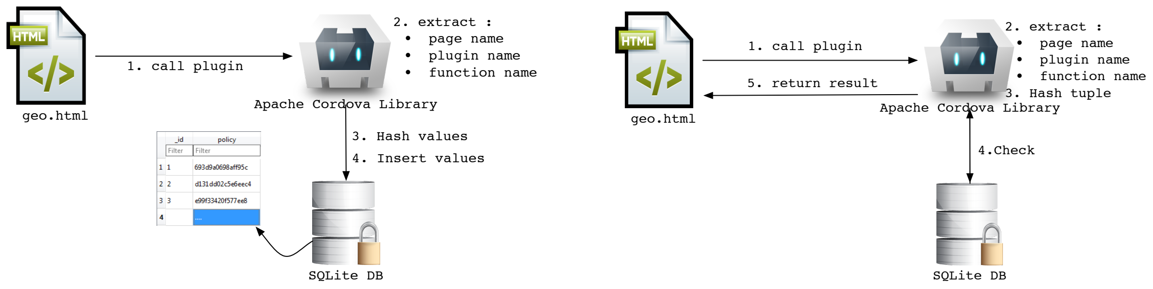
To address the third challenge, we updated the implementation of `PluginManager` plugin mapping logic to accommodate the plugin access rules and to grant access only to the pages specified in the policy stored in the database. When the `PluginManager` receives a plugin execution request, it retrieves the address of the currently loaded page by executing the WebView's `getUrl()` method and the Action requested from the plugin. Then it checks if the stored policy database contains a rule that grants the requested function from the plugin to the currently requested page. If access is allowed, then the `PluginManager` forwards the request to the corresponding plugin, else it will not forward the request and will generate a `PluginResult` that includes an illegal access permission message, which will throw an exception in the JavaScript. Figure 8 shows the reference monitor code inserted in the `PluginManager` execute method.

```
if(onCreate(checkPluginRule(service, action, currentPage))){
      //send request to selected plugin
} else {
      Status status = PluginResult.Status.ILLEGAL_ACCESS_EXCEPTION;
      PluginResult cr = new PluginResult(status);
      app.sendPluginResult(cr, callbackId);
      return true;
}
```

Figure 8: Proposed PluginManager Rule Check

5. Market Hybrid Apps Analysis

In this section we discuss the app analysis performed on real PhoneGap apps to investigate the common patterns used by hybrid apps in terms of file structure, plugins, permissions, and security settings. The PhoneGap website [6] hosts a repository linking to different app markets for apps developed using the PhoneGap framework. We have downloaded all free PhoneGap Apps (662 apps) that were build over



(a) Build Stage (b) Enforce Stage

Figure 7: Build/Enforce Policy

Android from Google Play [5] and were recommended by PhoneGap on Jun/2013. The apkdownloader (v.1.8.2) was used to automate the download of the apps from Google Play. The apktool (v.2.0) was used to extract files from the downloaded apps which include `AndroidManifest.xml`, `config.xml` and all application files under `www` folder. We built a C++ tool to parse and extract information from the retrieved files, this information includes the app permissions, plugin declarations, plugin usages and other configuration details.

PhoneGap app architecture The choice to develop Single Page App (*SPA*) or Multi Page App has been always a decision that the developer needs to take even though that most Hybrid platforms consider using *SPA* a good practice for many reasons related to performance and simplicity. For each of the apps downloaded we computed the number of local HTML files (pages) used by the app and the similarity between the different pages based on their accessed plugins. Figure 9 shows that more than 58% of the scanned apps are composed of more than one page, which shows that apps tend to have multiple pages. The average number of pages per app was 12.2 pages.

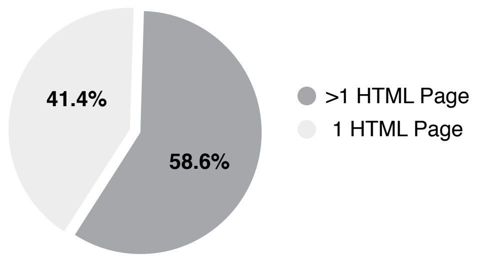


Figure 9: HTML File App Count

To investigate the similarity between the app pages based on their access plugins, we scanned the api calls in HTML/JS source for each page and generated a feature vector describing the plugins used in the page. The feature vector x for a page is a binary vector with $x_i = 1$ if plugin i was used and 0 otherwise. We computed the cosine similarity metric $sim(x, y) = \frac{x.y}{\|x\| * \|y\|}$ to compute the similarity between the pages in the same app. Figure 10 shows the average page similarity distribution for scanned apps, where as similarity of 0 implies pages used disjoint plugin sets and 1 means matching plugin sets. The overall average page similarity was 0.46. The majority of the apps have a similarity in the range $[0 - 0.5]$, which implies that different pages have different plugin usage requirements.

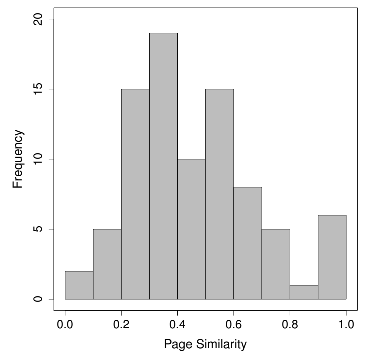


Figure 10: PhoneGap dataset page similarity distribution

PhoneGap Plugin Usage vs. Declaration. We scanned the api calls in HTML/JS source for each app and recorded the used plugins. In addition, we scanned the configuration files and recorded the declared plugins. Figure 11 shows the comparison between the number of plugin declarations and actual plugin usage. It can be concluded that there is a wide gap between what is declared and what actually gets used, for example 91.6% of the apps declare the GeoBroker plugin and only 8% of these apps actually use this plugin. This over declaration can be easily exploited by malicious scripts.

Access Origin Usage Patterns. Domain whitelisting is the current security model adopted by PhoneGap to control access to outside domains and subdomains. This model relies on the developer to provide the list of whitelisted domains. For example, developers could grant access to google.com by adding an `<access origin="http://google.com" />` to the whitelist. We extracted the access origin entries specified in all the apps in our dataset, and we categorized the access origin declarations into the following three categories: *open* or * which means app is open to any domain if `origin="*"` was specified as an allowed access origin, *local* if localhost is listed as an allowed access origin, and *specific* if a specific url was listed as a whitelisted access origin. Figure 12 shows the different access origin category com-

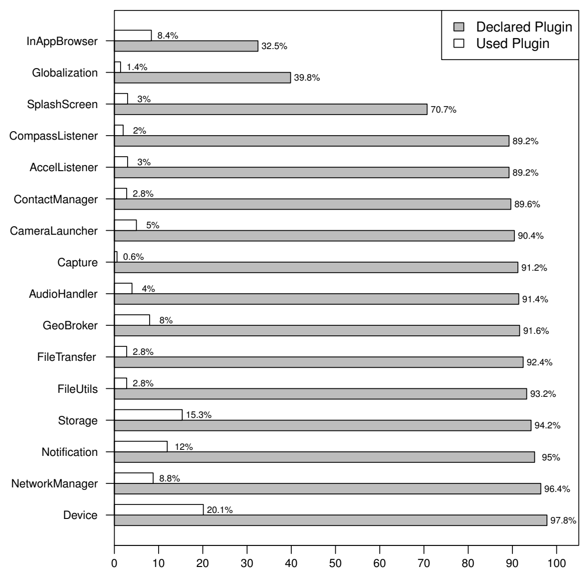


Figure 11: Plugin declaration vs plugin usage

binations and their statistics presented, note that the grey combinations show the risky settings, where 16.8% specified (*) access, 35.7% specified a (* *& local*) access, 0.8% specified (* *& specific*) access, and 5.7% specified (* *& local & specific*) access. Granting *open (*)* access is very risky as this allows access to any domain. These results highlight that 59% of the apps granted open access to any domain, which is an indication that the developers are not configuring their apps correctly and are relying on the nonsecure default PhoneGap settings. Apps granting open access to any external domain are subject to dynamic script loading from malicious sites.

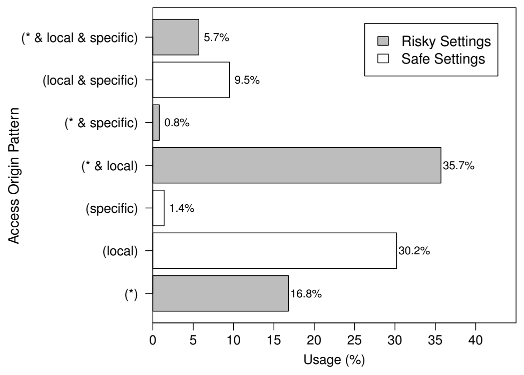


Figure 12: Access Origin usage distribution

6. Related Work

There has been several works in literature focusing on the design of access control models for Web-based apps in different contexts. Jin et al. [8] addressed the security problems encountered due the existence of bridges between PhoneGap and Android, those bridges break the protection that was already implemented in the WebView because they create holes in the sandbox of the WebView usage architecture. They study the reduction of the permission reachability without effecting business model so they designed a permission based access control model to control permission usage based on frames, it allow developers to assign different permissions to different frames. The set of permissions a frame is allowed to access is referred as the effective permissions. Their rules can be encoded in the HTML file or in the `AndroidManifest.xml` file. The policy enforcement performed by extending Android WebKit library to parse these attributes or extending the AndroidManifest parser. Native code is added to check for the rules before loading the page, hence; the existing reference Monitor is extended to check the effective permissions when an application tries to access protected resources. Their work is mainly focused on implementing a reference monitor in the Android Core, which is only applicable to Android and cannot be easily extended to other platforms for hybrid apps as it would require a major change in the smartphone's core libraries. In addition, the use of permissions as a base of these access rules is misleading to the developer as it would be much easier if access rules are based on plugins which the developer is familiar with. For example, the geolocation plugin requires the `ACCESS_COARSE_LOCATION` and `ACCESS_FINE_LOCATION`, it would be much easier for the developer to use the plugin name instead of specifying specific permissions.

Singh [12] has proposed an information flow model to control access in hybrid apps. Singh proposed approach enables developers to compose fine-grained, context-sensitive policies. Singh's model is mainly focusing on controlling the information flow between the different sources, this approach can be combined with our proposed approach to enforce more inter-component control.

Georgiev et al. [4] focuses on preserving the same origin policy (SOP) in hybrid apps. They demonstrated how hybrid frameworks do not properly compose the access control policies governing web code (HTML/JS) and local code (native code) highlighting that WebViews inside the app are not governed by same origin policy. They introduced the term *Fracking* which refers to any attack that allows foreign malicious Javascript to drill through the defense layers and gain unauthorized access to resources violating the same origin policy. They presented *NoFrack*, tool that enforces SOP through using a local DB (SQLite) to store whitelisted domains along with hashed id, they modified PhoneGap native code and the JS libraries to initialize each iframe accessing a domain with hashed id that will be later at run time to authenticate access any call to the bridge by using the id whenever a plugin call is executed.

Adappa et al. [1] address the security and privacy considerations of a mobile user while executing mobile mashups. They characterize the access control nature needed for mashups to access mobile device resources, and design an XACML based middleware to enable user control access of mobile features and integrate it into an existing mashup framework. Their middleware operates between the mashup framework and the mobile OS. The proposed policy captures many attributes of the app and user, and it is discussed that the framework is aware of the mashup and user's context. Their proposed approach introduces serious complexity issues for users when configuring the proposed policies, especially for complex apps that have many attributes. In addition, the proposed protection layer is dependent on the hosting mobile OS, which is not suitable for hybrid apps.

7. Discussion and Conclusion

Our proposed approach assumes that apps are composed of multiple pages and that there is variation in terms of the plugins requested by each page. However, we believe that the same proposed approach can be also used on *SPA* apps. In a single page app the object is still the plugin but the subject here is not the page name but the page content. The HTML content of the page being loaded at the time a plugin is executed is the state that can be used to identify the subject. The Cordova library can be modified using similar approaches, but instead of extracting the page name, it will extract the page fixed content, that is the layout related after removing all user data dependent HTML code. Android for example enables that through setting the `WebviewClient` built-in event `onPageFinished` to inject Javascript that calls `document.innerHTML`. This approach might cause some restriction on dynamicly updating the app UI, since it will generate different HTML content, nonetheless, this limitation can be avoided if the app went through the build stage. Build stage will refresh the data base with all possible interface states which will address this issue.

Ensuring the security of cross platform mobile apps is essential to their success given their vast popularity. In this paper we focused on the Apache Cordova library that is a common component used by most mobile hybrid frameworks to enable them access device native features. We presented its access control mechanisms and settings, and we presented the limitations of the current security settings. We proposed a framework that enables developers to automatically build an access control policy to control the plugins accessible to each page of the app and enforce these policy at app runtime. Given that the Apache Cordova framework whitelists all domains by default, it is important to carefully control device plugins exposure to the outside world. We implemented a prototype of our proposed approach. We analyzed a repository of 662 PhoneGap Android Apps and presented app page structure statistics, plugin declaration to usage statistics and access origin patterns. Hybrid apps will continue to address security and privacy issues, especially that it highly depends on dynamic loading which eliminates the chances of detecting malware behavior through conventional vetting process. We consider this work as a step towards understanding this ecosystem and adding more control on accessing device features without effecting the app business model.

Acknowledgments

This research was partially supported by grants from the National Science Foundation (NSF-CNS-0831360, NSF- CNS-1117411) and a Google Research Award.

References

[1] S. Adappa, V. Agarwal, S. Goyal, P. Kumaraguru, and S. Mittal. User controllable security and privacy for mobile mashups. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile '11, pages 35–40, New York, NY, USA, 2011. ACM.

[2] A. B. Bhavani. Cross-site scripting attacks on android webview. *CoRR*, abs/1304.7451, 2013.

[3] E. Chin and D. Wagner. Bifocals: Analyzing webview vulnerabilities in android applications. In *In Proc. of the 14th International Workshop on Information Security Applications (WISA)*, August 19-21 2013.

[4] M. Georgiev, S. Jana, and V. Shmatikov. Breaking and fixing origin-based access control in hybrid web/mobile application frameworks. In *In Proc. of 21st Annual Network and Distributed System Security Symposium (NDSS)*, Feb. 2014.

[5] Google. Google Play. `"https://play.google.com/store/apps"`, August 2013.

[6] P. Inc. PhoneGap Inc. `"http://www.phonegap.com/"`, February 2013.

[7] R. v. d. M. Janessa Rivera. Gartner Says by 2016, More Than 50 Percent of Mobile Apps Deployed Will be Hybrid. `"http://www.gartner.com/newsroom/id/2324917"`, February 2013.

[8] X. Jin, L. Wang, T. Luo, and W. Du. Fine-grained access control for html5-based mobile applications in android. In *In Proceedings of the 16th Information Security Conference*, 2013.

[9] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin. Attacks on webview in the android system. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 343–352. ACM, 2011.

[10] R. Mahesh Babu, M. 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*, pages 1–6. IET, 2012.

[11] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9): 1278–1308, 1975.

[12] K. Singh. Practical context-aware permission control for hybrid mobile applications. In *Research in Attacks, Intrusions, and Defenses*, volume 8145 of *Lecture Notes in Computer Science*, pages 307–327. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-41283-7.

[13] S. Xanthopoulos and S. Xinogalos. A comparative analysis of cross-platform development approaches for mobile applications. In *Proceedings of the 6th Balkan Conference in Informatics*, pages 213–220. ACM, 2013.