

SECURING MOBILE HYBRID APPLICATIONS THROUGH
CONFIGURATIONS - FIRST LINE OF DEFENSE

by

Abeer AlJarrah

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Computing and Information Systems

Charlotte

2018

Approved by:

Dr. Mohamed Shehab

Dr. Heather Lipford

Dr. Weichao Wang

Dr. Chuang Wang

ABSTRACT

ABEER ALJARRAH. Securing Mobile Hybrid Applications through Configurations
- First Line of Defense. (Under the direction of DR. MOHAMED SHEHAB)

Mobile hybrid apps have the potential to dominate the mobile and IoTs apps market. Cross-platform or hybrid apps are providing a promising development choice that appeals to a great body of developers. This development approach “wraps” standard web code (HTML, Javascript and CSS) into a thin native layer, enabling the same code base to run on several platforms. This approach also provides a mechanism to access device native sensors such as camera, geolocation and more, through Javascript code. As much as this seems innovative and promising, enabling web-code to access device native sensors is comparable to opening a can of worms in security terms. Most web-based vulnerabilities can be leveraged in mobile apps context which means amplified damage. Apache Cordova is an open source library that is a common component in many hybrid platforms including PhoneGap and IBM Worklight. Yet, it suffers several security limitations such as a coarse-grained access control model, risky defaults, and for many developers a non-trivial configuration process. Hybrid app development is an intricate task as is, not to mention configuring these apps securely. Given the increased popularity of the approach itself and the proven tendency of developers to use platform-provided default settings, this work aims to harden the middleware by implementing security mechanisms. Addressing security limitations on the platform/ middle-ware level reduce the cost of potential breaches significantly. In mobile hybrid apps context, we are focusing on one main app component; that is

the configurations. This work aims to provide different mechanisms to help developers by adopting configurations that are more aligned with the app requirements, and that implements the *Least Privilege* principle. Fine-grained and aligned configurations should help nullify several code injection attacks. To achieve this, we present 2 frameworks to implement a fine-grained plugin access model. The first one is a page-level and the second is more granular to enforce policies on a state-level.

In addition, we provide a tool that is mainly meant to incorporate the developer into the configuration process. We have implemented CORDOVACONFIG, an interactive web-based tool that is based on the state-level approach. Moreover, it provides more control to the developer and increases her awareness to the impact of risky settings. We have tested this tool, and our experiments demonstrate that it is a practical and a usable alternative for configuring hybrid apps.

We believe that fortifying hybrid mobile development process with functionalities that complies with security principles and involves the developer, is essential to enhance the quality and security of hybrid apps as a product. This is especially relevant with the current absence of proper supporting tools.

ACKNOWLEDGMENTS

I would like to express the deepest appreciation to my committee chair Professor Mohamed Shehab, who has the attitude and the substance of an inspiration: he continually and convincingly conveyed a spirit of brilliance in regard to research and scholarship, and an excitement in regard to teaching. Without his guidance and persistent help this dissertation would not have been possible.

I am grateful to all of those with whom I have had the pleasure to work during this and other related projects. Each of the members of my Dissertation Committee has provided me with personal and professional guidance and taught me a great deal about both scientific research and life in general.

Nobody has been more important to me in the pursuit of this project than the members of my family. I would like to thank my parents, whose love and guidance are with me in whatever I pursue. My brothers and sisters, whose unconditional love and support helped me bypass several obstacles. My UNCC friends whom I share with the same challenges and the same dreams.

I am especially indebted to Dr. Mary Lou, Chairman of the Department of SIS, who have been supportive of my career goals and who worked actively to provide me with the protected academic time to pursue those goals.

Finally, I would like to highlight that this work would not have been possible without the financial support of the Graduate Assistant Student Plan (GASP) awarded by the University of North Carolina at Charlotte.

TABLE OF CONTENTS

| | |
|--|------|
| LIST OF FIGURES | x |
| LIST OF TABLES | xiii |
| CHAPTER 1: INTRODUCTION | 1 |
| 1.1. Statement of Hypothesis and Approaches | 4 |
| 1.2. Summary of Contributions and Dissertation Organization | 5 |
| CHAPTER 2: PRELIMINARIES | 7 |
| 2.1. Mobile Development Approaches | 7 |
| 2.2. Apache Cordova Library | 11 |
| 2.3. Configurations of Cordova-Based Apps | 19 |
| 2.3.1. Configuration Items | 20 |
| 2.3.2. Configurations Evolution | 26 |
| 2.3.3. Configurations & Security Consideration for Hybrid Apps | 29 |
| 2.4. Threat Model | 30 |
| 2.4.1. Attack Form 1: Code Injection | 33 |
| 2.4.2. Attack Form 2: Compromised Third-Party Providers | 34 |
| 2.4.3. Attack Form 3: Apps Repackaging Attack | 35 |
| 2.4.4. Attack Form 4: Event Oriented Exploits - Return Oriented based Attack | 36 |
| 2.4.5. Examples of Malicious Impact on Hybrid Apps | 38 |
| 2.4.6. “Bad” Configurations Risks on Hybrid Apps | 39 |
| 2.5. HTML-5 Based App Development | 42 |

| | |
|--|-----|
| | vii |
| 2.6. Android WebView | 44 |
| CHAPTER 3: RELATED WORK | 49 |
| 3.1. Cordova Library Access Control Model | 49 |
| 3.2. Hybrid Apps Specific Attacks and Solutions | 50 |
| 3.3. Securing Apps by providing tooling support | 53 |
| 3.4. Security-By-Contract (SxC) on Mobile Code | 55 |
| CHAPTER 4: HYBRID APPS STATUS QUO: SECURITY & STATISTICS | 57 |
| 4.1. Market Analysis - 2014 | 57 |
| 4.1.1. Data Collection | 58 |
| 4.1.2. Results | 58 |
| 4.2. Market Analysis - 2017 | 62 |
| 4.2.1. Data Collection | 63 |
| 4.2.2. Results | 64 |
| 4.3. Cordova Common Vulnerabilities (CVEs) | 77 |
| CHAPTER 5: SECURING HYBRID APPS THROUGH THE APP CONFIGURATIONS | 81 |
| 5.1. Page Level Configuration Model | 81 |
| 5.2. Behavior-Based Configuration Model | 86 |
| 5.2.1. Plugin Access Policy | 88 |
| 5.2.2. Behavior State Modeling | 95 |
| 5.2.3. Performance Analysis | 99 |

| | |
|--|------|
| | viii |
| CHAPTER 6: CORDOVACONFIG: AUTOMATED TOOL FOR CONFIGURING HYBRID APPS | 103 |
| 6.1. Educational Goals | 108 |
| 6.2. App Behavior Synthesis Phase | 109 |
| 6.3. Generating Configurations & Permissions | 111 |
| CHAPTER 7: USER STUDY: HYBRID APPS DEVELOPERS PERCEPTIONS | 113 |
| 7.1. Case Study App | 114 |
| 7.2. Recruitment | 116 |
| 7.2.1. User Study Protocol | 116 |
| 7.3. Using CORDOVACONFIG | 118 |
| 7.4. Results | 119 |
| 7.4.1. Participants Demographics | 119 |
| 7.4.2. H1:CORDOVACONFIG & Configurations Understanding | 121 |
| 7.4.3. H2:CORDOVACONFIG and Developers Mental Model | 122 |
| 7.4.4. Q1: Developers' awareness of potential risks | 124 |
| 7.4.5. Q2: Perception of the benefits of CORDOVACONFIG | 127 |
| 7.4.6. Tool Usability | 129 |
| 7.5. Limitations | 131 |
| CHAPTER 8: CONCLUSION | 132 |
| REFERENCES | 135 |
| APPENDIX A: CORDOVACONFIG Screen Shots | 141 |
| APPENDIX B: Survey Questions A & B | 146 |

LIST OF FIGURES

| | |
|---|----|
| FIGURE 1: Different tools, languages and distribution channels associated with leading mobile operating systems[53] | 8 |
| FIGURE 2: Mobile Development Approaches[59] | 10 |
| FIGURE 3: Hybrid Platforms position in the Life-cycle of Technology Adoption [70] | 12 |
| FIGURE 4: PhoneGap config.xml file | 16 |
| FIGURE 5: Plugin access control execution flow | 17 |
| FIGURE 6: Mobile Hybrid App Architecture | 20 |
| FIGURE 7: Default Configurations as of version 8.x | 21 |
| FIGURE 8: Possible plugin access abuse | 31 |
| FIGURE 9: Attack Forms 1 & 2 | 33 |
| FIGURE 10: Hybrid Apps Repackaging Attack | 36 |
| FIGURE 11: EOE Attack Model [76] | 37 |
| FIGURE 12: Targetting Device Plugins/Data | 47 |
| FIGURE 13: Targeting App Behavior | 48 |
| FIGURE 14: HTML File App Count | 59 |
| FIGURE 15: PhoneGap dataset page similarity distribution | 60 |
| FIGURE 16: Plugin declaration vs plugin usage | 61 |
| FIGURE 17: Access Origin usage distribution | 62 |
| FIGURE 18: Access Rules Usage | 65 |
| FIGURE 19: Network Resource Access: Platform Provided Settings | 67 |
| FIGURE 20: Network Resource Access: Custom Settings | 68 |

| | |
|---|-----|
| FIGURE 21: Intents White-list Settings | 68 |
| FIGURE 22: Navigation White-list Settings | 69 |
| FIGURE 23: Core Plugins APIs | 71 |
| FIGURE 24: Custom Plugins APIs | 72 |
| FIGURE 25: Platform Native Permissions Usage | 74 |
| FIGURE 26: Content Security Policy Usage | 76 |
| FIGURE 27: Example Multi-Page App and Access Models | 82 |
| FIGURE 28: Policy stage in config file | 84 |
| FIGURE 29: Build/Enforce Policy | 85 |
| FIGURE 30: Proposed PluginManager Rule Check | 86 |
| FIGURE 31: Three-Stage Behavior Policy | 87 |
| FIGURE 32: Abstraction of Syntax Tree Representing States | 91 |
| FIGURE 33: Plugin Access Policy | 94 |
| FIGURE 34: App State Configurations | 97 |
| FIGURE 35: Check Redirection | 99 |
| FIGURE 36: <i>Enforce</i> Time vs # States | 100 |
| FIGURE 37: <code>exec()</code> with Plugin Type | 101 |
| FIGURE 38: CORDOVACONFIG Work Flow | 107 |
| FIGURE 39: Explaining scanned configurations meaning/impact | 108 |
| FIGURE 40: Plugin accesses and OS permissions captured <i>per</i> state | 109 |
| FIGURE 41: State Transition Example | 110 |
| FIGURE 42: App interaction with external entities | 111 |

| | |
|---|-----|
| FIGURE 43: Generated Configurations | 112 |
| FIGURE 44: Employee Directory App | 115 |
| FIGURE 45: Common Coding Practices followed by participants | 120 |
| FIGURE 46: Developers' Configuration Understanding Scores | 122 |
| FIGURE 47: Developers' Mental Model Change Scores | 124 |
| FIGURE 48: Perceived Implications of having unaligned plugins settings | 125 |
| FIGURE 49: Perceived Implications of having unaligned Network Access settings | 126 |
| FIGURE 50: Perceived Benefits of CORDOVACONFIG | 128 |
| FIGURE 51: SUS Scale with red Arrow indicating CORDOVACONFIG Score | 130 |
| FIGURE 52: SUS Distribution | 131 |
| FIGURE 53: Start Screen | 141 |
| FIGURE 54: Current Configuration Analysis-Part1 | 141 |
| FIGURE 55: Current Configuration Analysis-Part2 | 142 |
| FIGURE 56: Current Configuration Analysis-Part3 | 142 |
| FIGURE 57: Plugin access captured for a state | 143 |
| FIGURE 58: Plugin access captured for a state | 143 |
| FIGURE 59: App transition diagram | 144 |
| FIGURE 60: App interaction with external components | 144 |
| FIGURE 61: Generated Configurations | 145 |

LIST OF TABLES

| | |
|---|-----|
| TABLE 1: Mobile Development Approaches Summary | 11 |
| TABLE 2: Whitelist Configurations | 22 |
| TABLE 3: Configuration Items | 23 |
| TABLE 4: Cordova Configuration History Summary | 27 |
| TABLE 5: Policies Break Down | 77 |
| TABLE 6: Cordova Security Vulnerabilities[28] | 78 |
| TABLE 7: Configurations Issues and impact on apps' security | 104 |
| TABLE 8: Variables Measurement Methodology | 114 |
| TABLE 9: Participants Development Experience | 119 |

CHAPTER 1: INTRODUCTION

Smartphones, wearable devices, and the system of Internet of Things (IoTs) are eventually becoming mainstream, replacing desktops not only as personal gadgets but also as workplace tools. This will increase the demand for Enterprise mobile apps, which is expected to outstrip the available development capacity 5 to 1 according to Gartner [65]. Enterprises are striving to maintain apps that can run on different devices with low cost. This explains enterprises' increasing interest in adopting a Mobile Hybrid development approach [50] since this approach satisfies the business need to leverage mobile applications across many platforms. It also provides the capability to use mobile device features using standard web technology. Adopting this approach eases administrative tasks in the Bring Your Own Device (BYOD) environment; moreover, it enables using a single code base in many platforms, which drastically reduces the cost by targeting the whole market and discarding platforms' market fragmentation [13][71]. From a developer's point of view, research [44][13] shows that *Fragmentation* is the major challenge mobile developers are facing as they have to deal with multiple mobile platforms. The fact that advocates for using cross-platform apps more appealing to mobile developers more than other mobile development approaches. Not to mention that they show great potential for rapid development of high-fidelity prototypes of mobile apps[13].

Mobile Hybrid development uses standard web technology (HTML5, JavaScript, CSS)

“wrapped” into a thin native layer, which enables the app to run on different platforms. This approach is a mix of the other two approaches on Mobile development, namely *Native* and *Web-Based*. Native apps’ advantage of accessing device native features is leveraged in Hybrid apps through the native wrapper. At the same time, the same code base is used for different platforms, making it possible to ship to a wider customer base in a low cost.

As this Hybrid approach continues to experience acceptance within the developers’ community, organizations must think seriously about how key changes in this latest paradigm will require them to shift their application security practices for Web and Mobile Apps. Given the fact that Hybrid Apps are basically web-based; most web-based vulnerabilities are shipped to smartphone devices rather than a confined web browser window. The effect of any attack on a hybrid app is amplified because the app has the ability to access device native features through a bridge implemented by the hybrid platform middleware [41][42]. This requires reconsidering the current approaches to be more conformant to the new paradigm.

Cordova Library [3] is a middle-ware which is a common component in many popular hybrid platforms such as PhoneGap, IBM Worklight, App Builder, Sencha, Monaca, and Appery.io. This component is the real enabler of connecting the two worlds (Web and Native) inside a hybrid app. However, the library endures limitations that raise security concerns such as unsafe defaults and a coarse-grained configuration model. Moreover, the library has vague configuration documentation, especially those related to security settings [11].

Cordova-Based Apps Configurations In the latest release of the ten most critical

web application security risks, OWASP [21] indicates that security misconfiguration is a serious issue given that defaults are usually not secure. This is congruent with the results we have from scanning 662 Cordova based apps, which uncovered a serious issue in that regard. We have found that 81% of the apps have misaligned configurations, meaning they have API declarations that are not actually used. Moreover, 58% have risky settings such as allowing loading resources from *any* domain. This is a result of using the default settings provided by the library which is not necessarily secure, especially the early versions of Cordova library[3] because a default Content Security Policy (CSP) was not provided by default. This also can be attributed to developers' tendency to overlook security settings and focus only on the functional part of the app.

In this work, we aim to investigate different approaches of identifying and fixing weaknesses in hybrid mobile middlewares. We aim to identify weak access models adopted and to help developers to consider security aspects of their apps as early as possible during the life-cycle of the app.

We focus on the configuration scheme adopted by Cordova library, we highlight its weaknesses and propose two approaches that aim to provide more fine-grained and aligned models. We address a page-level access control model that defines plugin access to be per page rather than per the whole app.

We also address a *behavior-based* approach to securely configure hybrid apps. This approach provides the benefit of securing against novel attacks, yet adding minimal effort on the developer's side, by automatically generating a fine grained access policy to govern app behavior in terms of plugin access and state transition. Our proposed

approach captures state-plugin access rules and state transition rules through monitoring the app behavior while running in a controlled environment and then enforcing these rules when the app is released to the market. This approach aims to provide behavior based configuration on app state level, minimizing the effort on the developer side.

Moreover, we focus on enabling the developer by providing tooling and educational support. We implement an interactive tool `CORDOVACONFIG` that is built to enable developers to control the configurations based on monitored behavior. The aim of this tool is not only to involve developers in the configuration process but also to increase their awareness and knowledge. Hence, we conduct a user study to measure the tool effectiveness in increasing developers' awareness in that regard. The proposed approach is well-suited for the security of hybrid apps, especially that the security of this technology is in a relatively immature state compared to the conventional mobile development approaches. It is highly expected that there are still several attack forms and implications that are not yet discovered. Particularly, because the hybrid software stack adds new complexity by bridging external web code to internal smartphone sensors creating new security concerns that are specific to this category.

1.1 Statement of Hypothesis and Approaches

This presented research here hypothesizes that:

- The Cordova-Configuration scheme addresses several potential security breaches and there is a need for fine-grained Cordova-based app configurations to reduce

the attack surface on device-resources

- Developers have minimal knowledge on configuring apps securely
- Providing developers with automated tools to help configure their applications will reduce the attack surface and will increase developer awareness in that regard.
- Given the relative novelty of this app category, we hypothesize that monitoring apps' behavior in a *healthy* environment and enforcing it later can be used to control critical resource accesses and protect the app against the risks of hybrid app-specific attacks, regardless of the attack type.

1.2 Summary of Contributions and Dissertation Organization

The contributions of this securing hybrid apps research are as follows:

- We explore the Hybrid Apps ecosystem in terms of internal implementation of Cordova library, market share, developers' practices and common vulnerabilities
- We conduct a large-scale market apps scan to better understand developers practices in terms of coding and configurations.
- We identify attack scenarios applicable to hybrid mobile applications focusing on the impact of these attacks on users' privacy and security
- We demonstrate an approach to reduce the attack surface on hybrid apps by implementing a fine-grained configuration scheme

- We demonstrate an automated behavior-based configurations tool to generate aligned configurations that can be used to control app behavior.
- We implement CORDOVACONFIG; a tool that provides an interactive configuration process giving developers more control to configure hybrid apps
- We conduct a user study to measure the effectiveness of CORDOVA CONFIG in increasing developers' awareness in Cordova configurations

The remainder of this dissertation is organized as follows: Chapter 2 discusses preliminary work explaining Mobile development approaches and how they relate to each other. Then, explaining the Apache Cordova internal implementation and then discussing HTML5 coding practices and finally a brief description of Android Web View since Android is the platform used. Chapter 3 reviews current applications, and explores related work. In Chapter 4 we explore the current status-quo of hybrid apps through investigating developers' practices in developing and configuring hybrid apps and exploring the threat model related to hybrid. Then, in Chapter 5 we explain two approaches for securing hybrid apps through configurations. In Chapter 5 we present the tool CORDOVACONFIG, which offer interactive configuration process and in Chapter 7 we discuss the results of the user study conducted on the tool. Lastly, Chapter 8 discusses future work and concluding remarks.

CHAPTER 2: PRELIMINARIES

The 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 different settings to control and setup the communication channel between the embedded web browser component and the hosting native app. In this chapter, we first explain different mobile app development approaches including the trendy hybrid apps development and compare them in several aspects. Second, we focus on the Apache Cordova library and explain its architecture and configuration model. Then, we explain in detail configuration items and their impact on an app. After that, we demonstrate the threat model on hybrid apps and how they are related to poor configurations. Finally, we highlight HTML5 development practices and explain Android **WebView**'s to help convey the design decisions we made and the implementation of our tools.

2.1 Mobile Development Approaches

Native Apps: Binary executable files designed for vendor devices. They can access all APIs made available by the OS vendor. SDKs are platform-specific. Each mobile OS comes with its own unique tools and GUI tools. Examples of the main Mobile

OSs are shown in Figure 1 Once the app is installed, it interacts with the underlying

| | Apple iOS | Android | Blackberry OS | Windows Phone |
|------------------|---------------------|-------------------|-------------------------|--|
| Languages | Objective-C, C, C++ | Java(some C, C++) | Java | C#, VB.NET and more |
| Tools | Xcode | Android SDK | BB Java Eclipse Plug-in | Visual Studio, Windows Phone development tools |
| Packaging format | .app | .apk | .cod | .xap |
| App stores | Apple App Store | Google Play | Blackberry App World | Windows Phone Marketplace |

Figure 1: Different tools, languages and distribution channels associated with leading mobile operating systems[53]

operating system through proprietary API calls that the OS exposes. These are divided into 2 categories: Low-level APIs and high-level APIs. Through low-level API calls, the app can interact directly with the touchscreen or keyboard, render graphics, connect to networks, process audio received from the microphone, receive images and video from the camera, access the GPS etc. Higher level services include processes like browsing the web, managing the calendar, contacts, photo album, and the ability to send and receive phone calls,.. etc. Native apps have full use of all functionalities that modern mobile devices have to offer. On the other hand, natives apps cannot be used for other platforms. They are also expensive to develop in terms of time and development skills.

Web-Based Apps Modern mobile devices consist of powerful browsers that support many new HTML5 capabilities, Cascading Style Sheets 3 (CSS3) and advanced JavaScript. With recent advancements on this front, HTML5 signals the transition of this technology from a page-definition language into a powerful development standard for rich, browser-based applications. Mobile web apps are a very promising trend.

To capitalize on this trend and help developers build the client-side UI, a growing number of JavaScript toolkits have been created, such as `dojox.mobile`, Sencha Touch and jQuery Mobile, which generate user interfaces that are comparable in appearance to native apps.

One of the most prominent advantages of a web app is its multi- platform support and low cost of development. Unlike native apps, which are independent executables that interface directly with the OS, web apps run within the browser. The browser is itself a native app that has direct access to the OS APIs, but only a limited number of these APIs are exposed to the web apps that run inside it. While native apps have full access to the device, many features are only partially available to web apps or not available at all. Although this is expected to change in the future with advancements in HTML, these capabilities are not available for today's mobile users.

Hybrid Apps The hybrid approach combines native development with web technology. Using this approach, developers write significant portions of their application in cross-platform web technologies, while maintaining direct access to native APIs when required. The native portion of the app can be developed independently, but some solutions in the market provide this type of a native container as part of their product, thus empowering the developer with the means to create an advanced application that utilizes all the device features using nothing but web languages. In some cases, a solution will allow the developer to use any native knowledge he or she might have to customize the native container in accordance with the unique needs of the organization. The web portion of the app can be either a web page that resides on a server or a set of HTML, JavaScript, CSS and media files, packaged into the

application code and stored locally on the device. Both approaches carry advantages and limitations. HTML code that is hosted on a server enables developers to introduce minor updates to the app without going through the process of submission and approval that some app stores require. Unfortunately, this approach eliminates any off line availability, as the content is not accessible when the device is not connected to the network. On the other hand, packaging the web code into the application itself can enhance performance and accessibility, but does not accept remote updates. The best of both worlds can be achieved by combining the two approaches. Such a system is designed to host the HTML resources on a web server for flexibility, yet cache them locally on the mobile device for performance.

Figure 2 summarizes the different approaches. The native approach excels in perfor-

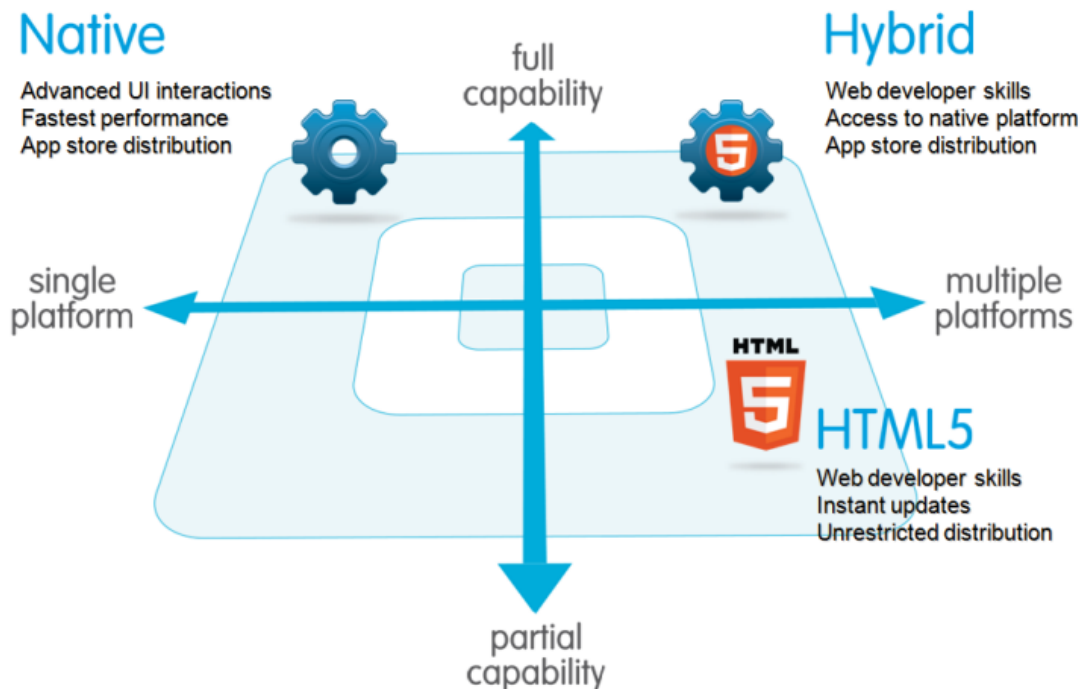


Figure 2: Mobile Development Approaches[59]

Table 1: Mobile Development Approaches Summary

| Feature | Native | Web-Based | Hybrid |
|-----------------------------------|---------------|------------------|---------------|
| Development language | Native only | Web only | Mixed |
| Code portability and optimization | None | High | High |
| Access device-specific features | High | Low | Medium |
| Leverage existing knowledge | Low | High | High |
| Advanced graphics | High | Medium | Medium |
| Upgrade flexibility | Low | High | Medium |
| Installation experience | High | Medium | High |

mance and device access, but suffers in cost and updates. The web approach is much simpler, less expensive and easier to update, but is currently limited in functionality and cannot achieve the exceptional level of user experience that can be obtained using native API calls. The hybrid approach provides a middle ground which, in many situations, is the best of both worlds, especially if the developer is targeting multiple operating systems. Table 1 summarizes the features of each approach.

2.2 Apache Cordova Library

The Apache Cordova Library is an open source library that enables building mobile apps with HTML, CSS and JS. It targets multiple platforms with one code base. It supports 8 platforms including Android, iOS, Windows, and Blackberry. Applications execute within “wrappers” targeted to each platform and rely on standards-compliant API bindings to access each device’s sensors, data, and network status[3]. Diverse platforms use this library, including PhoneGap, Ionic, Visual Studio and Intel XDK [3]. The PhoneGap platform in particular is gaining popularity over other options for many reasons, including flexibility, straightforward architecture and ease of use not to mention the relatively low consumption of memory, CPU, and power [29][52]. Hybrid

development, or the approach of building native apps using Web technologies, has gone through its fair share of highs and lows. But, despite high-profile abandonments from companies such as Facebook and LinkedIn, hybrid development continues to be used by a substantial number of developers. According to a recent article [70], the adoption of Mobile Hybrid platforms is steadily increasing and is enjoying a stable stage in the technology adoption life-cycle. Developers are still overwhelmed by the need to support a growing number of platforms. Hybrid development, and its single codebase approach, resonated with developers, particularly Web developers who salivated at the opportunity to write native iOS and Android apps with the technologies they already knew. Despite its relatively recent presence, Cordova-based apps re-

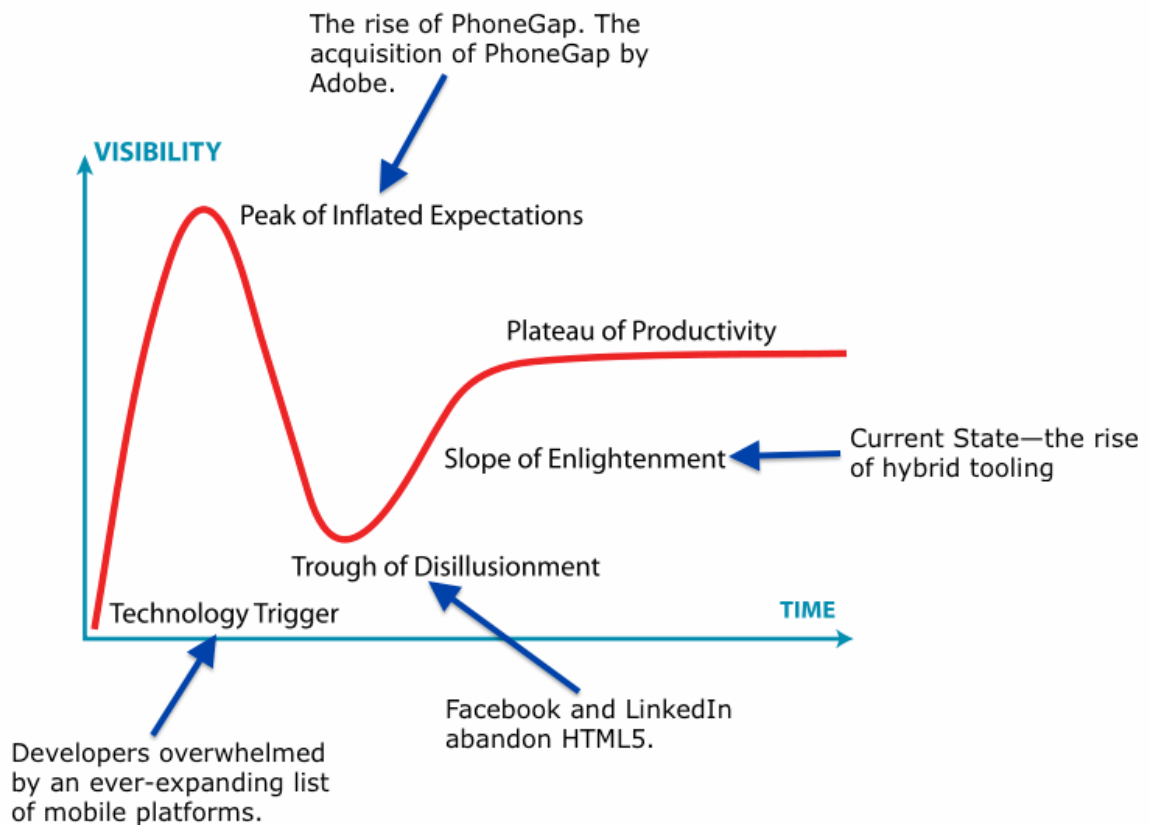


Figure 3: Hybrid Platforms position in the Life- cycle of Technology Adoption [70]

cent statistics in the Google Play store shows that it constitutes 5.84% of the market share. Some apps are able to attract 10,000,000+ customers [15]. Business, Medical and Finance tops the list of Cordova-based app categories. Moreover, observing the platform's increasing popularity, recent research work suggested augmenting the Cordova library with wider functionalities such as a voice agent and embedded WebRTC [63][40][71].

Cordova applications are implemented as a browser-based `WebView` within the native mobile platform. A Cordova plugin is add-on code that provides a JavaScript interface to native components. It allows the app to use native device capabilities beyond what is available to pure web apps such as the camera, contacts and geolocation. At the time of writing this dissertation, Cordova supported 3627 plugins that are open source and available to the public[3].

Cordova also implements a bridge to connect the two worlds (Web and Native) together. `CordovaWebView` adds a Javascript Interface using the method `AddJavascriptInterface()` which enables the `WebView`'s internal Javascript code to call the native method `CordovaPlugin.exec()`. The `exec()` function is the entry point to any plugin on the native side of the app. After the native side executes the plugin, the result is saved in a queue of Javascript messages that are injected back to the `WebView` using `loadUrl()`. Moreover, the Cordova JS side might trigger the `WebView` to display an alert dialog, confirmation or prompt. This is implemented by customizing the event handlers in `WebChromeClient`. This object is mainly to model how `WebView` should react to Javascript dialogs, favicons, titles, and the progress through overriding methods.

Recently (as of *version 4.0* [11]), a default Content Security Policy (CSP) is included in the startup *index.html* generated by the library. According to the library documentation, CSP is meant to control network requests such as images and XML HTTP Requests (XHRs) made via WebView directly which cannot be controlled using Network Request Whitelisting. On Android, for instance, the network request whitelist is not able to filter all types of requests; such as <video >and WebSockets. So, in addition to the whitelist, a CSP via the <meta >tag is required on all pages. However, CSP is not effective nor practical for two main reasons. The first reason is that CSP is *not* mandatory in Cordova-apps. Meaning, if a developer copied her web-based files into the *www* folder inside the app, replacing the default *index.html* file provided by the library, there will be no CSP yet the app runs with no errors. Second, CSP documentation is not simple rather than informative in terms of potential risks on the app if CSP is not set properly. This may not encourage developers to use it, especially those whose ultimate priority is getting the app to work. Such developers are not expected to go further into considering security policies in their code.

There are several cross-platform frameworks that implement the Apache Cordova library, but PhoneGap is actually a distribution of Apache Cordova. For the sake of simplicity, we discuss Android Apps generated through PhoneGap as an example to explain how the library works. The same context applies with the other platforms that use the Cordova library. As discussed earlier, PhoneGap uses the Apache Cordova engine as a middleware that provides APIs to establish communication channels between the native code and JavaScript. 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. The PhoneGap framework is extensible; it is possible for developers to include their own customized third-party plugins which require the developer to define both native plugin libraries and JavaScript interfaces. 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 4 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 app activity that hosts the embedded WebView, it 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 (JavaScript) components included in a PhoneGap project that are needed for communication between native side and web side:

- **ExposedJsApi** (Native): The 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 the `addJavascriptInterface`.

```

<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 4: PhoneGap config.xml file

- **PhoneGap Client Library (Client):** The JavaScript library (`cordova.js` or `phonegap.js`) 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):** This is the central component in apps operation, it is responsible for the initialization of the different plugins included in the app, 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.

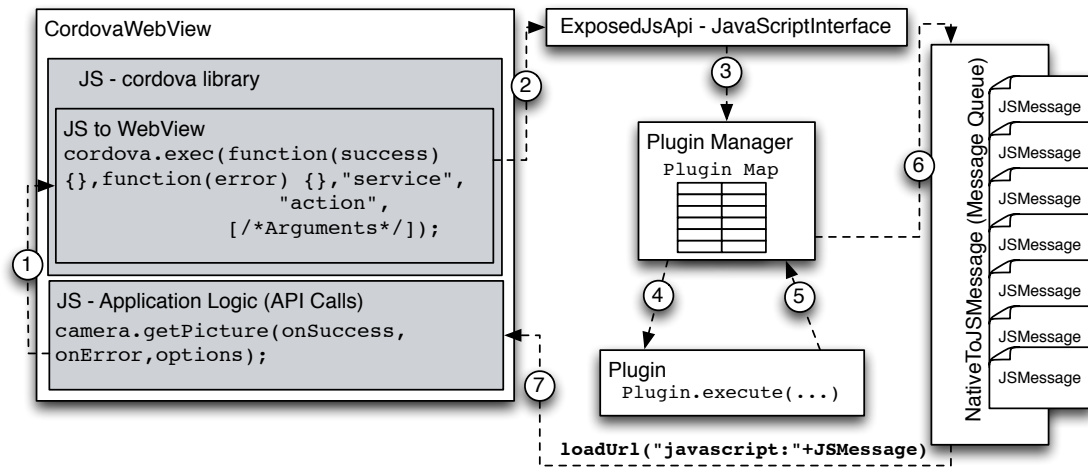


Figure 5: Plugin access control execution flow

Figure 5 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 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 (before 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 called method calls the `PluginManger.exec()` method 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 4, 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` (Steps 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 then 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. 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 Privacy Guide which contains recommended policies to be used by developers when using plugins, especially those accessing private information like the camera, contacts and geolocation information. Despite of the controls provided by

PhoneGap to control plugin access, it is still possible to compromise apps and bypass these checks as is discussed in the following section.

2.3 Configurations of Cordova-Based Apps

A Cordova-based app, regardless of the platform, uses one global configuration source file, namely `config.xml`. This XML based file located under `/res/xml` path and contains several settings that control app behavior. The configurations are mainly updated through the Cordova command line interface (CLI) or by simply editing the file. Configurations are parsed and the content is then translated into a set of features (see Fig. 6). Those features form the policy of accessing the device native APIs and define the app local and external interactions with web domains. For instance, in a cordova Android-based app, certain components in the library, mainly `PluginManager` and `CordovaActivity` are responsible for enforcing these features. Default configurations specify global properties like name, description, author, preferences, and domain whitelisting specifications. Adding a plugin API changes the configuration to include a declaration of that plugin by adding a `<feature>` tag. CLI also adds the required system permissions to the native side to permit using the device native features. The default `config.xml`, according to the the version (*version 8.x*) only includes the `Whitelist` feature. This feature manifests the security model provided by the library through providing the developer with configuration items that control the app interactions with external entities such as URLs and other installed apps using the domain whitelisting model. Table 2 explains the specifications provided by this plugin indicating the defaults. Domain whitelist configurations can be

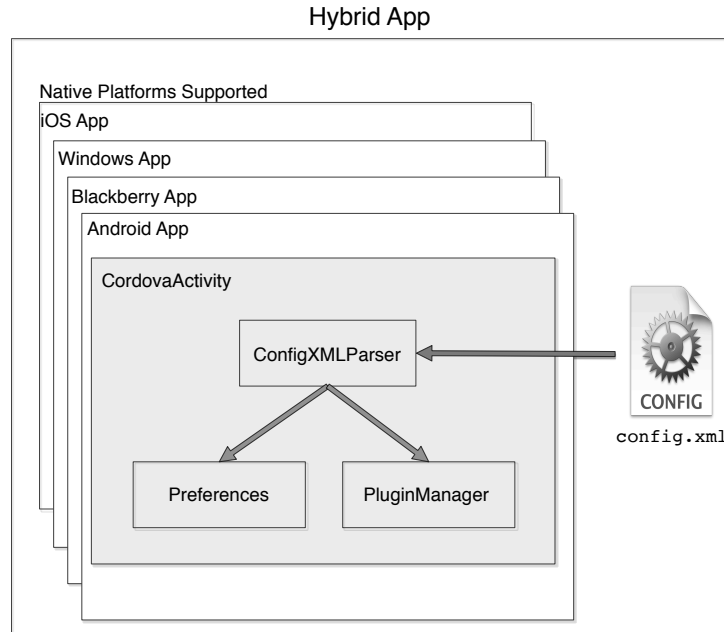


Figure 6: Mobile Hybrid App Architecture

categorized into Network Request Whitelisting, Navigation Whitelisting, and Intent Whitelisting.

2.3.1 Configuration Items

Default configuration settings are shown in Fig. 7. Configurations are mostly common among different platforms, however there exist certain configurations and options that are specific to certain platforms. We summarize the description of the common configuration items in Table 3 as described in their manual [4].

We categorize the configuration items into two categories based on their functionality and impact:

- Descriptive configurations: configurations related to providing meta-data about the app and the author. Examples include `widget`, `name`, `description`, `author`, and `platform`.

```

<?xml version='1.0' encoding='utf-8'?>
<widget id="com.example.hello" version="1.0.0" xmlns="http://www.w3.org/ns/widgets"
xmlns:cdv="http://cordova.apache.org/ns/1.0">
  <name>HelloWorld</name>
  <description>
    A sample Apache Cordova application that responds to the deviceready event.
  </description>
  <author email="dev@cordova.apache.org" href="http://cordova.io">
    Apache Cordova Team
  </author>
  <content src="index.html" />
  <plugin name="cordova-plugin-whitelist" spec="1" />
  <access origin="*" />
  <allow-intent href="http://*/*" />
  <allow-intent href="https://*/*" />
  <allow-intent href="tel:*" />
  <allow-intent href="sms:*" />
  <allow-intent href="mailto:*" />
  <allow-intent href="geo:*" />
  <platform name="android">
    <allow-intent href="market:*" />
  </platform>
  <platform name="ios">
    <allow-intent href="itms:*" />
    <allow-intent href="itms-apps:*" />
  </platform>
</widget>

```

Figure 7: Default Configurations as of version 8.x

Table 2: Whitelist Configurations

| Syntax | Default | Meaning |
|--|---------|---|
| <code>access origin="*"</code> | ✓ | Allow accessing resources from all domains |
| <code>allow-navigation href="http(s)://*/*"</code> | | Allow links to all http(s) URLs to be loaded |
| <code>allow-navigation href="data:*"</code> | | Allow data of all formats to be passed into the WebView |
| <code>allow-intent href="http://*/*"</code> | ✓ | Allow all http links to web pages to open in a browser |
| <code>allow-intent href="https://*/*"</code> | ✓ | Allow all https links to web pages to open in a browser |
| <code>allow-intent href="sms:*"</code> | ✓ | Allow SMS links to open messaging app |
| <code>allow-intent href="geo:*"</code> | ✓ | Allow geo: links to open maps |
| <code>allow-intent href="tel:*"</code> | ✓ | Allow tel: links to open the dialer |
| <code>allow-intent href="market:*"</code> | ✓ | Allow market: links to open Google Play Store |

- Behavior-control configurations: configurations related to controlling app access to device APIs, interacting with external domains and installed apps, and the starting page. Examples include `content`, `access`, `allow-navigation`, `allow-intent`, and `feature`.

Descriptive configurations are simple and self-explanatory as explained in Table 3.

We focus on the app’s behavior-control configurations as they can have a substantial impact on the app security and behavior.

We explain the main configuration parts:

Table 3: Configuration Items

| Item | Description |
|-------------------------------|---|
| <code>widget</code> | Root element of the <code>config.xml</code> document |
| <code>name</code> | Specifies the app's formal name, as it appears on the device's home screen and within app-store interfaces. |
| <code>description</code> | Specifies metadata that may appear within app-store listings. |
| <code>author</code> | Specifies contact information that may appear within app-store listings such as author's email and website. |
| <code>content</code> | Defines the app's starting page in the top-level web assets directory. The default value is <code>index.html</code> , which also appears in a project's top-level <code>www</code> directory. |
| <code>access</code> | Defines the set of external domains the app is allowed to communicate with. The default value allows it to access any server. |
| <code>allow-navigation</code> | Controls which URLs the WebView itself can be navigated to. Applies to top-level navigations only. |
| <code>allow-intent</code> | Controls which URLs the app is allowed to ask the system to open. By default, no external URLs are allowed. |
| <code>preference</code> | Sets various options as pairs of name/value attributes. |
| <code>feature</code> | Specifies the device APIs the app is allowed to use. Examples include camera, geolocation, ..etc. |
| <code>platform</code> | Specifies preferences or other elements specific to a particular platform. |

Plugins APIs: Hybrid apps can access device native features such as `CAMERA`, through a Javascript API provided by the library. A developer needs to include the plugin name in the configurations using the tags `<feature>` or `<plugin>`, depending on the version. Developers can use command lines to add plugins which also updates the native side automatically to include the required system permissions and the native code needed to use the plugin. For instance, including the `CAMERA` API in an Android app changes the `AndroidManifest` file to include the permission: `"android.permission.CAMERA"` and the config file to include a declaration of the camera plugin. Earlier versions of the library include by default a set of core plugins in the configurations along with all required system permissions.

Domain White-listing for Accessing Network Resources: this is the security mechanism that controls access to external domains over which the app has no control. Cordova provides a configurable security policy to define which external sites can be accessed. By default, new apps are configured to allow access to any domain. The library “recommends” that developers customize the white-list to allow access to specific domains and subdomains before moving the app to production. For Android, Cordova’s security policy (as of its 4.0 release) is extensible via a plugin interface `cordova-plugin-whitelist`, as it provides better security and configurability than earlier versions of Cordova. The library configurations use `<access>` element within the app’s `config.xml` file to enable network access to specific domains. For example, to access domain `XYZ.com`, access should be configured as : `<access origin="http://XYZ.com">`. The default value for this configuration item is `“*”` which allows access to any domain. It is important to mention that there is a limita-

tion in the enforcement of this item, as described in the library documentation. The issue is that white-listing alone can not block network redirects from a white-listed remote website (i.e. http or https) to a non-whitelisted website which jeopardizes the app to be vulnerable to several attacks. Examples include injection attacks, XSS, and untrusted JavaScript code inclusion. To overcome this, the library recommends using a content security policy (CSP) on a page level to mitigate redirects to non-whitelisted websites for webviews that support CSP.

Content Security Policy: A fundamental component of the platform security model. Its main purpose is to mitigate XSS and injection attacks. It controls which network requests (images, XHRs, etc) are allowed to be made (via webview directly) on a page level. CSP is set by including a `<meta>` tag listing the rules of content download. A CSP consists of a set of policy directives and corresponding values. Examples of directives include `default-src`, `script-src`, and `img-src`. These are responsible for control of default content, script and images respectively. `default-src` is a default directive that applies in case other directives are not specified. Policies help controlling allowed URL sources, executing inline code, and enabling `eval()` function.

In Cordova, a default CSP policy: (1) disables `eval()` and in-line scripts style; (2) allows network requests of types CSS, AJAX, and frame; and (3) allows only local code execution. CSP is added to the library starting in version 5.0.0 and is included by default in the generated `index.html` file.

Domain White-listing for Intents: Like any native app, a hybrid app may interact with other system components including other installed apps. In Android, this is handled by `Intents` which is designed to manage inter-application communi-

cation. Hybrid apps may access other intents such as dialer, email or messages via URIs and may also pass data to these intents. Thus, cordova library provides rules via `allow-intent` to govern what intents can be called. As is demonstrated in Fig. 7, default configurations allow interaction with the following apps: dialer, messages, maps, market, and the browser. This white-list applies to calls via hyperlinks and the method `window.open()`.

Domain White-listing for Navigation: controls which URLs the WebView itself can be navigated to. Applies to top-level navigations only. On Android, it also applies to iframes for non-http(s) schemes. By default, navigations are allowed only to local files. To allow other URLs, the item `<allow-navigation>` can be used to set the list of URLs.

2.3.2 Configurations Evolution

Initially in version 1.5.0, the library supported 6 platforms (Android, Blackberry, iOS, Symbian, WebOS and Windows Phone) and offered APIs to access 13 device resources (Accelerometer, Camera, Capture, Compass, Connection, Contacts, Device, Events, File, Geolocation, Media, Notification, and Storage). The most recent version of the library (at the time of writing this dissertation) is 8.x. This version supports 7 platforms (adding WP8), ships with 16 core APIs in addition to supporting a repository of 2895 plugins developed and shared by the community¹. We track and report the changes on the library starting from version 1.5.0 till the most recent version in Table 4. We only highlight the changes related to configurations and security in ad-

¹<https://cordova.apache.org/plugins/>

Table 4: Cordova Configuration History Summary

| Version | Change(s) |
|---------|--|
| 1.5.0 | <ul style="list-style-type: none"> • Core plugins are included by default • Plugins declarations in file plugin.xml. • No security model. |
| 1.8.0 | <ul style="list-style-type: none"> • Domain whit-listing is introduced. • Documentation added a section about white-listing guide. • Access rules in file cordova.xml. |
| 1.9.0 | <ul style="list-style-type: none"> • CLI is shipped with the library. |
| 2.0.0 | <ul style="list-style-type: none"> • Plugin developers guide is introduced. • Access rules in file config.xml |
| 2.8.0 | <ul style="list-style-type: none"> • Documentation added a privacy guide. |
| 3.0.0 | <ul style="list-style-type: none"> • No plugins are included by default. • CLI enables adding plugins. • Documentation added configuration reference. • White-list plugin is introduced. |
| 3.5.0 | <ul style="list-style-type: none"> • Documentation added security guide. • Content-Security Policy (CSP) is added. |
| 5.0.0 | <ul style="list-style-type: none"> • White-list plugin introduces <code><allow-navigation></code> • White-list plugin introduces <code><allow-intent></code> |

dition to any security awareness effort made by the library. In version 1.5.0, all core plugins are included by default in the configurations. This would also effect the native side of the app because it would be configured to include the system permissions needed to use all the included APIs. An Android app, for instance, would have 15 system permissions by default, including CAMERA, ACCESS_COARSE_LOCATION, and INTERNET. At that time, there was no consideration of any security principle such as controlling app interaction with external domains. It wasn't until the release of version 1.8.0.that the concept "Domain White-listing" was introduced to support controlling access to external domains [5]. At that time, domain white-listing was implemented for 3 platforms (Android, iOS, and Blackberry). Access rules to outside domains are specified in an XML format using the element `<access>`. It was set to

`origin="*" which allows access to any server by default. These rules are stated in a file named cordova.xml. Later in version 1.9.0 the library presented a Command Line Interface (CLI) which is a standard set of command-line tools. The purpose is to simplify interaction with the library and make it easier to develop cross-platform applications. It was supported for 3 platforms, Android, iOS, and Blackberry. In version 2.0.0 the library started to encourage developers to develop their own plugins, hence the library released a plugin development guide in their manual; explaining the architecture of the Cordova API on both sides JavaScript and native. In version 2.8.0, the library dedicated a section in their documentation for privacy. The privacy guide at the time encouraged developers to follow practices to preserve users' privacy such as, having a privacy policy, avoiding collecting users' information, and giving users more control on allowing collecting information and accessing device APIs. Version 3.0.0 had a major change compared to previous versions. First, plugins are not included by default. Second, adding plugins is supported commands through CLI. Third, a domain white-listing policy is implemented in the file config.xml.`

Not until the release of version 3.5.0, was a security guide added in the documentation [9]. The guide addressed security issues that are discussed in research [48][8][7]. It focused on several security breaches including bringing awareness to issues related to setting access to specific servers rather than allowing access to all servers.

To provide more modularity and extensibility to the security model, the library provided a white-list plugin for Android and iOS in version 5.0.0. This plugin provides better security and configurability compared to earlier versions of Cordova as it added more security related configurations such as `<allow-navigation>` and

<allow-intent>. In addition to that, the library implemented a Content Security Policy (CSP) to allow more control of content downloads at a page level.

2.3.3 Configurations & Security Consideration for Hybrid Apps

Efforts to make the library more secure (driven either by the research community or the library itself) are evident. Yet, the library still suffers security limitations in their configuration model.

2.3.3.1 Coarse-Grained

Configuration rules are enforced on the app regardless of the context. The policy fails to address context-aware details such as location and time. Moreover, configurations are global to the whole app. The app consists of one or more pages or states. Each of which requires different permissions based on the app behavior in a specific state. However, the configurations grant permissions to the app as a whole.

2.3.3.2 Risky defaults

Default settings of the configurations are liberal. Most default values grant non-restricted access. Values, such as “*” indicating non-restricted access, are common. For instance, if we examined the default settings for domain white-listing for network resource access is set to “*” which allows access to any server - in the absence of CSP. One consequence is the capability of including remote JavaScript from remote untrusted servers into the app. Moreover, intents white-listing default settings allow access to several built-in apps including dialer, SMS, maps, and browser.

Having risky defaults is a serious issue given the high probability that developers are likely to keep default values as is.

2.3.3.3 Inefficient Security Model

The content security policy was added to control access on page level and to address limitations of domain white-listing. However, this security model is not as efficient because its usage is not enforced. In other words, the absence of a CSP does not prevent finding the app, it would only log a warning message. This may not necessarily draw developers' attention to include it at all. Not having CSP and keeping the network resource setting to default, shall expose device resources to an untrusted server. This jeopardizes the security of the app to enable injection attacks.

2.4 Threat Model

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 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, and hence, increases the attack surface of the app. In figure 8(a), we demonstrate a possible attack that can be launched on an app developed using Cordova library before version 3.1.0. Some of the plugins like Accelerometer

don't need any permission to execute, which enables an attacker to access this plugin easily by calling it through javascript. Later versions of Cordova use a minimalistic

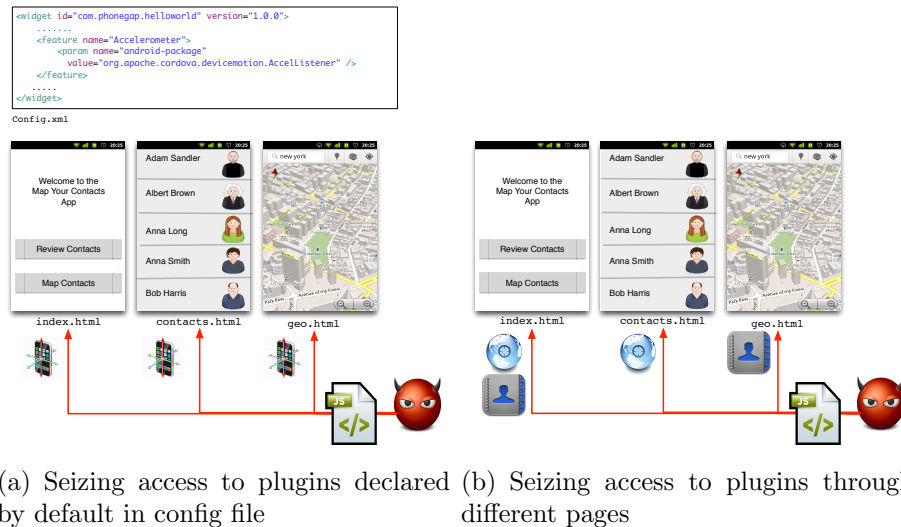


Figure 8: Possible plugin access abuse

approach since by default the app doesn't have any plugin. The developer can use the help of Cordova Command-Line Interface (CLI) to add device level features which in turn configures the project accordingly. In both cases, plugin access is set on the app level, i.e. declaring a plugin means it can be accessed through any local or dynamic javascript file if the permissions requirements of the plugin are fulfilled. This addresses the possibility of malicious scripts to execute plugins through any page, see figure 8(b) which demonstrates how an attacker can access the geolocation plugin through the contacts page - that is meant to access only the contacts plugin, and the same applies to the rest of the pages in the app. As the previous threats were due to the current implementation of Cordova security access model, other threats might arise from 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 introduce several vulnerabilities [49] 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 4. The wildcard `<access origin="*" />` allows access to an external resource. In addition, it is possible to allow access to resources from specific domains, for example, `<access origin="https://www.google.com" />`.

There are many threats [17, 49, 24] associated with using WebViews in mobile apps. The `loadUrl` and `addJavascriptInterface` methods can be easily used to introduce back channels to give access to malicious apps [49]. For example, `loadURL` can also be abused 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. The Cordova framework is based on using browser-based components. 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 are selected.

The first two subsections explain attack scenarios applicable to hybrid apps. The ultimate goal of these attacks is to compromise a hybrid app to host a malicious code. The third subsection provides examples of malicious code and its effect on hybrid apps.

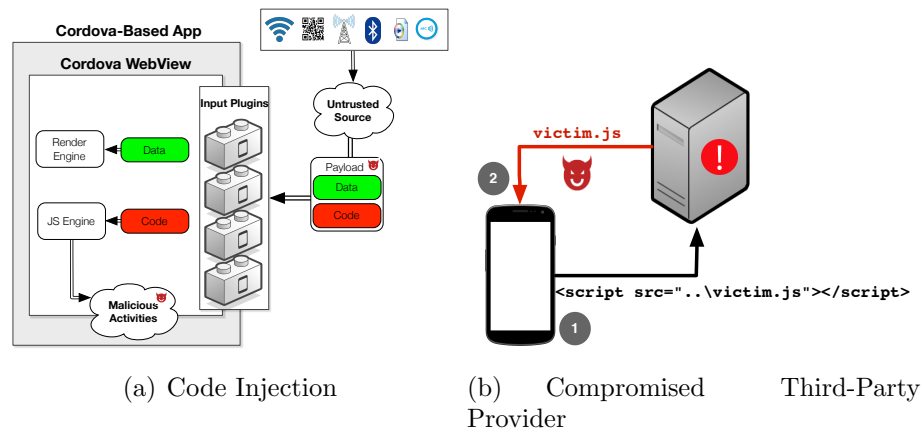


Figure 9: Attack Forms 1 & 2

2.4.1 Attack Form 1: Code Injection

Given that JS is subject to code injection and that hybrid apps use standard web technology, attackers can abuse smart phone specific features such as a camera; as new windows for code injection. The essence of this model stems from two basic facts:

- Data and code can be mixed and the mixed code can be triggered by a standard JS engine inside `CordovaWebView`.
- Smart phones by nature, provides broader interaction surface with the outer world compared to PCs, through device native features such as camera.

Cordova input plugins can be abused as channels to inject infected payloads of data

into the app [41]. Figure.9(a) describes the basic premise of these attacks. Untrusted data input resources such as WiFi access points, Bluetooth, Bar-codes, QR images and MP3 files can embed malicious code that can be triggered inside the infected app through the JS engine.

Recent work [37] have also discussed Web-to-Application injection attacks. Where malicious code injection can happen through passing malicious data to `Intent` through a link call. The target of such an attack is installed apps on the device abusing the bridge enabled to launch installed apps such as dialer and maps. One compromise scenario would be invoking a `BROWSABLE` intent passing a malicious `url` as a parameter.

2.4.2 Attack Form 2: Compromised Third-Party Providers

An important feature of JavaScript is the ability to combine many libraries from local and remote resources into the same page. Developers include remote resources for many reasons, such as decreased latency, increased parallelism, and better caching. Mobile Hybrid Apps developers, in particular, strive to create cross-platform apps with a native look, high performance, and rich functionality. Thus, UI specialized libraries (jQuery, Ionic, Framework7, Mobile Angular UI), Tracking libraries (Google Analytics), Social Integration libraries (Facebook) or Ad libraries (AdMob) are very common in cross-platform apps. Ideally, developers should include JS code from trustworthy providers, yet even these providers are not immune to attacks.

Figure.9(b) depicts how a compromised remote server can actuate malicious code execution inside a benign app through the inclusion of a malicious remote JS file.

Although developers trust that remote providers will not abuse the power bestowed upon them, the amount of damage that can be caused by compromising these remote servers is substantial. Remote JS code has the same privilege as the local code. Thus, it can access not only app resources such as data but also native features through the bridge provided by the hosting hybrid platform. This amplifies the damage that can be caused by this vulnerability which makes the provider of the library an interesting target for cyber-criminals. Attackers may also use client apps that do not follow security tips in regard to input sanitization to include malicious code through File inclusion vulnerability. *File inclusion vulnerability* is a common vulnerability in web application context. It allows an attacker to include a file, usually through a script on the web server. The vulnerability occurs due to the use of user-supplied input without proper validation. This can lead to something as minimal as outputting the contents of the file or more serious events such as native plugin code execution on the device. For instance, the developer intended only `file1.js` and `file2.js` to be used as input options. But it is possible to include code from other files as anyone can insert arbitrary values for the `PARAM` parameter. This line for instance `/vulnerable.php?PARAM=http://evil.example.com/webshell.js?` includes a remotely hosted file containing a malicious code.

2.4.3 Attack Form 3: Apps Repackaging Attack

Repackaging apps is an old threat that applies to all apps, and hybrid apps are not an exception. Attackers can compromise hybrid apps easily by repackaging the app, then adding/modifying the web-based code and then repackage it again with the new

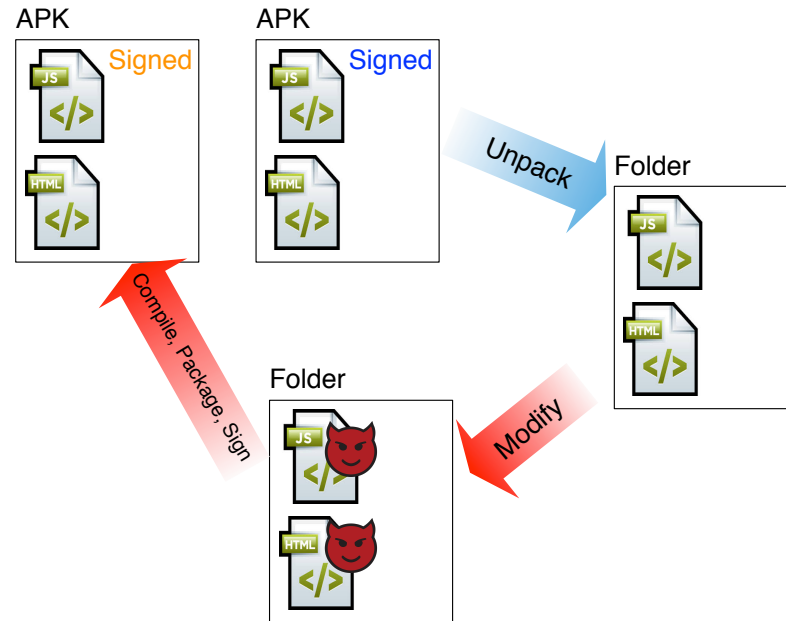


Figure 10: Hybrid Apps Repackaging Attack

code, see Figure 10. In fact, repackaging hybrid apps is much easier than repackaging Android apps because web-based code (javascript and HTML) remains as a plain text after unpacking. Unlike native Java code which is decompiled into Dalvik bytecode instead. Attackers can target popular apps of this category and inject malicious code. Users who install apps from untrusted resources will be infected.

2.4.4 Attack Form 4: Event Oriented Exploits - Return Oriented based Attack

A recent work [76] has discussed the applicability and severity of event-oriented exploits (EOE) on Android hybrid apps. This line of attacks belongs to the renowned return-oriented programming exploits. The stem of this attack is to gain control by abusing the call stack of a program. By carefully chaining together a sequence of call statements, an attacker can call arbitrary operations that would not execute in the normal control flow. In the case of hybrid apps, the fact that event handlers on the

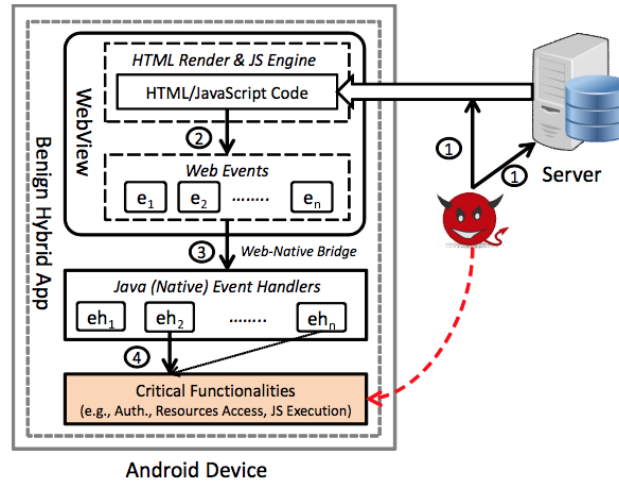


Figure 11: EOE Attack Model [76]

native side can be called from the web side, made this exploit feasible, see Figure 11. The internal critical functionalities can be utilized by triggering the associated web event and feeding it the proper input. For example feeding the webview the URI “`sdk://c1.c2?args:...&callback=..`” would trigger the native components `c1` and `c2` on the native side.

Specifically, attackers may abuse the combination of the functions: Webview’s `loadUrl` and WebviewClient’s `shouldOverrideUrlLoading` to create an implicit control flow. EOE has multiple advantages. First, it doesn’t require any extra permissions, i.e. the malicious code injected fully inherits the permissions granted to the app. Second, EOE doesn’t require malicious payloads. Instead, the functionalities contained in the event handlers are utilized. Moreover, compared to code injection attacks, EOE doesn’t require the code to be triggered nor the bridge to be enabled. This means, even having proper configurations, permissions and tight CSP will not prevent the exploit from happening.

2.4.5 Examples of Malicious Impact on Hybrid Apps

Malicious code damage can take several forms, such as data manipulation, privacy leakage, Denial of Service (DoS), or changing app behavior. The impact of the attack and the way the attack is taken differ based on the target of the attack. For example: **Target Device/User Data:** Hybrid platforms provide a bridge to connect to device sensors, thus malicious JS code may target the device sensors accessing private data such as tracking the location of a specific device as shown in Figure. (12). The malicious code in this example posts the compromised device geolocation coordinates every 3 seconds to a remote server.

Target App Behavior A recent market scan showed that Finance, Banking, and Health tops the market share of Cordova based apps [15], thus, compromising the app itself is a valid concern, particularly in these sectors. This can be carried out by malicious actions like malicious redirect, injecting un-wanted content like advertising, adult content, or manipulating the routing logic to bypass a mandatory check such as log-in or payment page. The Cordova `InAppbrowser` plugin provides the functionality of redirect from one URL to another. By injecting code such as that shown in Figure. (13), it does not only redirect to another URL, but also external code will be loaded into the application. The parameter `_self` means that `CordovaWebView` will host the external webpage not the system browser. Moreover, this plugin enables JS code injection using the method `executeScript()` method which enables executing a JS code. Although, this JS code injected by `InAppBrowser` cannot access Cordova APIs, at least it can manipulate the app behavior. This can successfully run using

any URL if configurations are kept on default settings. Previous work [41][22][20] suggested static JS analysis to detect malicious code; however, attackers may obfuscate their code, making it more challenging to be detected, especially if it was dynamically injected. JS provides many obfuscation libraries, one simple example using the function `escape()`.

Successfully redirecting the app to other pages has serious implications on the user experience, especially if the app is critical such as banking and shopping apps. Symantec declares Malicious JavaScript Redirection a severe vulnerability that could pose a serious security threat to the device[66]. Current default configurations shown in Table 2 are very loose, especially that the default network request white-listing `<access origin="*" />` doesn't block any access request if a CSP is not included in the page. This gives more privilege to the malicious code over the app and the device.

2.4.6 “Bad” Configurations Risks on Hybrid Apps

App configuration is a contract between the app and the system. The file `config.xml` content states what external domains can access the app, what external apps can be launched from the app, what plugins the app can use, and what domains can be navigated from *inside* the app's `WebView`. Secure configurations serve as a first defense line against potential attacks. Previous work on securing hybrid apps [41][22][20] have been focusing only on Javascript (JS) as a source of problem and solution. The approaches suggested so far are revolving around static and dynamic JS analysis to detect malicious code injection. Configurations, however, have been missing from the literature even though it is a basic component of the app security. The damage from

any malicious code execution can be voided by properly configuring the app. For example, an injection attack trying to access a camera is ineffective if the app is not configured to include the camera plugin in the first place.

As previously shown, default configurations in Table 2 are loose and coarse-grained. We demonstrate the impact of *bad* configurations that may result from either keeping default settings or having *relaxed* configurations

2.4.6.1 Network Request Whitelist

This configuration item controls which network requests are allowed to be made via Cordova native hooks. The default setting, `<access origin="*" />`, does not block any access request. The default index page generated by the library contains Content Security Policy (CSP) to control which network requests are directly allowed to be made via `WebView`. The default CSP allows only local URLs. CSP is enforced through the white-list plugin, which is added to the app by default. CSP is represented inside `<meta>` tags on the top of the HTML file. While including CSP is a secure practice that is encouraged by the library, not including one results only in showing a warning message. Controlling network requests is merely dependent on the possibility of the developer using a default generated index page that contains a default CSP, or the possibility of her adding a CSP to her customized pages. Having no CSP enforced and keeping default configurations may jeopardize the app to external resource loading/injection from *any* URL (1st row in Table 2).

2.4.6.2 Navigation Whitelist

This item controls which URLs the `WebView` itself can be navigated to. This setting applies to top-level navigations only. By default, it navigates to local files because, as shown in rows 2,3 in Table 2, these settings are not the default. To white-list specific URLs, configuration should include the corresponding value, for example, `allow-navigation href="http://example.com"`. However, as mentioned earlier, this applies only to top-level navigation, which means that any redirect from `example.com` to any other URL may not be in conformance with the specified setting. We argue the need of additional configuration settings to control all URLs being loaded into `WebView`. This helps not only protect against malicious URLs loading but also maintains the app behavior against any injection attack trying to bypass certain states in the app, such as authentication.

2.4.6.3 Intents Whitelist

This item controls which URLs the app is allowed to ask the system to open (through hyperlinks and `window.open()`). This includes Built-in Apps, such as Messaging, Dialler, Maps, Mail, and Browser. Default settings allow calling *all* the mentioned apps as shown in rows 4-9 in Table 2. Including these apps in the configuration file allows malicious injected code to launch these apps, not to mention passing values to them using crafted URIs. In fact, Cordova versions before 3.5 suffered from a vulnerability (CVE-2014-3502) [27] that allows remote attackers to open and send data to arbitrary applications via a URL with a crafted URI scheme for an Android intent. Hence, we argue that keeping the default settings as is will increase the attack

surface through this channel.

2.4.6.4 Declaring Plugins/Features

Early versions of Cordova configuration (before 3.1), used to include, by default, a set of 16 declarations of native APIs or plugins, including Camera, Geolocation, and Contacts. This default setting assumes that the app needs to use all the declared plugins, which is not necessarily true. Previous work [61] demonstrated the gap between the plugin declaration in the configuration file and the actual plugin usage in the code. Even if a developer included only the plugins needed by the app, these declarations assume that these plugins are used by every page/state in the app, which also creates a window for attack. We argue that a plugin declaration should be tied with the app state/page that needs to access the plugin rather than being tied to the whole app as one unit.

Most developers consider security a non-functional requirement. Thus, they are less likely to change default configurations. Even for developers who consider themselves security-sensitive, there is still a possibility that they might lack the knowledge to do so. This outlook is backed by a study [75] that reveals the existence of a disconnect between developers conceptual understanding of security and their attitudes regarding their personal responsibility and practices for software security.

2.5 HTML-5 Based App Development

HTML-5, Javascript and CSS have opened the door for platform-independent UI development, since it provides a rich UI experience not to mention support for several mobility features such as offline web storage, canvas drawing, and CSS3. Consider-

ations specific to mobile development must be taken into account to help provide the user with a full experience of a mobile app with good performance, given the limited processing capacity of mobile devices. Bandwidth is a major concern; thus, minimizing server requests and payload size while providing a rich UI is a priority. To achieve this, mobile HTML-5 based apps implement most of the logic on the client-side, leaving the server side for authentication and data access. This trend mandates applying strict mechanisms to keep client side Javascript as layered, modular and object-oriented as possible to help manage and maintain the code.

Single Page Applications, or (*SPA*), is a web-based app that fits into a single page with the goal of providing a more fluid user experience, akin to a desktop application. Developers are highly encouraged to follow this approach when developing mobile apps for several reasons such as improved performance, fast navigation between many views/pages, and less download content/network bandwidth in order to achieve a more native like experience.

SPA attempts to reduce the total number of pages that a user must load down to one. JavaScript routers provide a method for tracking user state and loading required resources, as needed, without requiring a URL change or page reload using *Hashbangs*. As the user navigates, the library changes the hashbang (#!) in the URL to denote their current location. Hashbangs are usually an alphanumeric word that represents certain action. It might also contain a parameter which can be a digit (for instance index in a list or an array) or a letter. Examples of hashbangs looks like `#employee/1/changePhoto`, `#/users/list/5`, `#pages/about`. To support SPA to navigate properly, certain logic should be implemented to act as a proxy between user

actions and the app engine:

(1) Routing: When the window hash changes, logic must apply to load the correct state from global states.

```
document.addEventListener('hashchange',function(e){  
    // swipe views here  
});
```

(2) Markup Management: Views rendered by manipulating the DOM variable. The current state decides how markups are updated.

```
if(current_view){  
    // insert before current view  
}  
else{  
    // insert to the content wrapper  
}
```

Manipulation of the Document Object Model (DOM) determines the view visible to the user. Gmail, Twitter, and Facebook are examples of SPAs. The urge to adopt this approach for hybrid apps is increasing. This also complies with our apps scan result, as approximately 60% of the apps pool have one html file.

2.6 Android WebView

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 work we will focus on

the 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, it includes methods to navigate forward/backward through history, to zoom in and out, to perform text searches in addition to many other methods [14]. Besides its ability to render the page, the WebView also 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.

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

The code 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. 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.

```
//Native Java Side
class JsObject{
public String save(String data) {
//save data
return value;
}
}

webview.addJavascriptInterface(new JsObject(),"injectedObject") ;

//Javascript Side
var value= injectedObject.save("data");
```

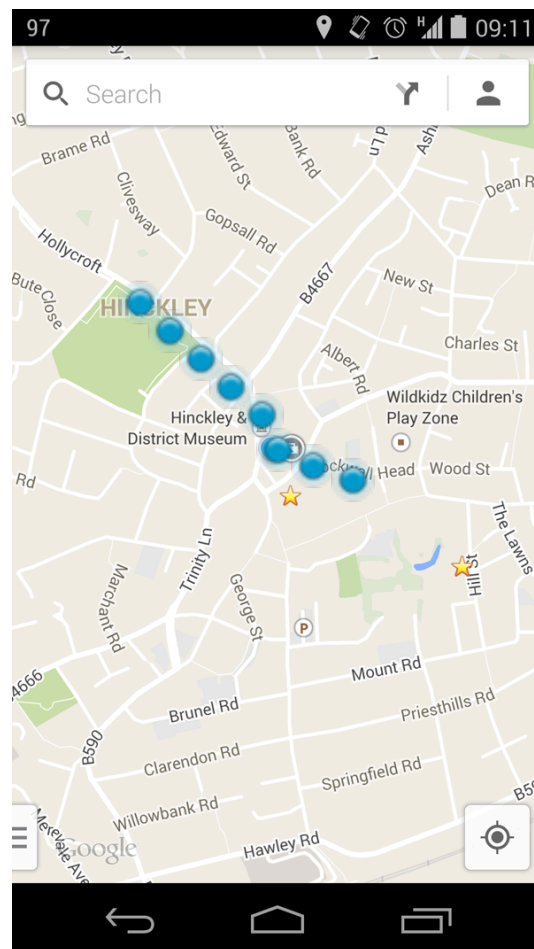
The code demonstrates how a native object is injected into the WebView using the `addJavascriptInterface` method.

```

1 <script>
2   var refreshID = setInterval(function(){
3     navigator.geolocation.watchPosition(
4     function(loc) {
5       $.post('someurl',format(loc),function({});
6       });}, 3000);
7   function format(loc){
8     var data= {'lat':loc.coords.latitude,
9               'long':loc.coords.longitude} ;
10    return JSON.stringify(data);
11  }
12</script>

```

(a) Access Device Data

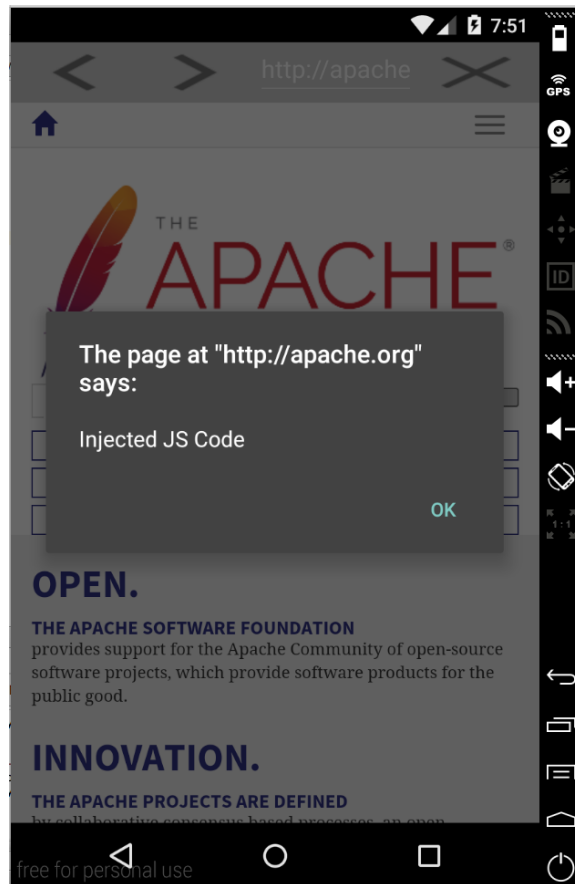


(b) Geolocation Data Exposed

Figure 12: Targetting Device Plugins/Data

```
1 <script>
2   var ref = cordova.InAppBrowser.open
3     ('http://www.someurl.com', '_self');
4   ref.addEventListener('loadstop',
5     function(){
6     ref.executeScript(
7       {code:"alert('Injected JS Code')"}
8     );
9   });
10 </script>
```

(a) Change App Behavior



(b) Infected App

Figure 13: Targeting App Behavior

CHAPTER 3: RELATED WORK

3.1 Cordova Library Access Control Model

Early work related to hybrid apps security [43][62][61][32] has focused on the coarse grained access control model offered by the library and on the risk associated with having a bridge implemented to connect native device resources with javascript (JS) code. Jine et al. and Singh et al. [43][62] suggested a finer-grained access model to control the native side permissions associated with external resources. MobileIFC [62] proposed to include a context-aware policy to control access to resources on both sides, native and web. The work suggested splitting JS code into confined chunks that are activated based on certain conditions. While Jin et al [43] have discussed how the privilege granted to Cordova-based Webviews breaks the existing protection mechanism provided by Android Webviews, they have proposed a fine-grained access control model on the *frame* level to be associated with native side permissions. Later, Kudo et al. [46] presented a run time access control model where the user needs to approve plugins access at run time. This mainly depends on a users' judgment and knowledge of what needs to be accessed and by which parties. Early work [32] have focused on the inefficacy of the Same Origin Policy principle (SOP) implementation and enforcement in Cordova library and proposed a change on both native and JS side of the library to extend origin based access control to local resources outside the

web browser.

Phu et al. [54] recommended HybridGuard, a principal-based stateful policy enforcement framework that enables developers to define policies and specify principal-based permissions.

While the line of this work aims to have a fine-grained access control model to limit API access by external JS code; however, as opposed to previous solutions, this work does not require developers to set a policy nor split JS into chunks. Our tool requires no effort on the developer side nor changing JS coding structure, yet provides a state level policy on plugins access and app behavior.

3.2 Hybrid Apps Specific Attacks and Solutions

Despite the relative newness of the technology, mobile hybrid apps security has gained a lot of attention. Code injection attacks viability of Cordova-based apps are discussed in several research papers [41][22][42]. In their work Jin et al [41][42] have explained how XSS attacks can be mounted on mobile devices using sensors such as Contact, SMS, Barcode, and MP3. These can be abused to serve as channels for receiving and spreading malicious code. The attack is based on the ability to mix code and data on web technology which can be used to inject malicious code to exploit plugin calls. To handle these attacks, they have modified the Cordova plugin manager to include code that sanitizes JS calls from malicious strings. A followup work by Chen et al [22] discovered another attack channel that is based on the ability to inject malicious JS code through HTML text input fields. They have also developed a tool named DroidCIA which is a tool that parses HTML files along with the JS files to

fully analyze the mobile app and sanitize malicious API calls.

Kudo et al. [46] have discussed repackaging attacks and highlighted that it is much easier to apply this attack on hybrid apps since JS and HTML code remain in plain text format even after packaging. Solutions such as obfuscating JS code [35] is discussed but there is still valid concerns on its effectiveness and applicability.

Several tools have been suggested by researchers that mainly aim to detect vulnerabilities using static and dynamic analysis of code [19][78][77][57][47]. Early work done by Brucker et al. [19] developed a static analysis tool for foreign language calls and detecting data flows in cordova based apps. Yang et al. [78] focused on sensitive methods in hybrid apps and studied their communication with the native side. They proposed a hybrid analysis tool that combines dynamic and static analysis. The main target here is to detect unsafe connections (HTTP or insecure HTTPS) that tries to invoke sensitive operations. Static analysis searches for sensitive functions and dynamic analysis investigates these functions' invocation from an unsecure connection. This work has several limitations such as its failure to intercept HTTPS connections and inability to perform complete dynamic analysis on apps that require access to credentials. Another similar work [47] presented HybriDroid, a static analysis framework for hybrid apps. The tool analyzes intercommunication between Android Java and Javascript.

Considering the dynamic nature of hybrid apps, dynamic taint tracking is proposed [64] to detect data leakage. The approach principally identifies data sources and sinks, taint sensitive data such as cookies, geolocation and offline storage, then tracks the channels and checks if the data is leaked. The main issue with this work is the weak

experimentations.

Yang et al. [77] presented BridgeScope that is a system for vetting JS bridge security among various implementations of Webviews in several platforms including Android and Mozilla. The tool uses static analysis, type taint and value analysis. The tool focuses on vetting sensitive API calls and tracing their call path. However, it suffers from limitations including not handling implicit data flows and low level libraries written in C/C++ which may lead to false negatives. Our approach can prevent and detect injection attacks based on observed app behavior. It can detect dynamic injection attacks which include obfuscated code which could be overlooked by conventional solutions such as JS sanitization and static JS analysis. Along the same lines, information flow analysis is used by Rizzo et al. [57] to evaluate the impact of code injection attacks against Webviews. Their approach relies on instrumenting apps and using an extension of the class Webview - “Babelview” to trace API calls.

A new attack vector on hybrid apps was presented by Yang et al. [76] that addressed Event-Oriented exploits on Android hybrid apps. The exploit is an extension of the well-known return oriented programming attacks. The basis of this exploit is the fact that hybrid apps allow event handlers defined in the native side to handle web events. Attackers can remotely access native device resources through event handlers in Webview without any permission or authentication. Critical internal functionalities can be utilized by triggering the associated web events and feeding it with proper inputs that follow the format : “`sdk://c1.c2?args=...&callback=...`”, where `c1` and `c2` are the native functions to be accessed, `args` are the function’s parameters and `callback` is a JavaScript function name to receive the execution result of the

native function. To detect such an exploit, the authors proposed EOEDroid, a tool that vets event handlers using selective symbolic execution and static code analysis. This direction of research, where attacks are discovered and a solution is suggested to counter the mentioned attack, is still in its infancy in regard to hybrid apps. There has been a recent effort [36] to systematize analyzing potential attacks and vulnerabilities in hybrid apps by investigating common security concerns based on standardized references such as OWASP and simulate them on hybrid apps. Yet, there is still an immense need to protect the app against zero-day attacks by enforcing specific rules that conform with app healthy behavior given the expected uncertainty of the attacks' channels.

3.3 Securing Apps by providing tooling support

Supporting developers to maintain privacy protective behavior when coding has been addressed in the literature in different aspects. Rebecca et al.[16] highlight key challenges developers face in following best practices related to privacy. Being a secondary concern, privacy is not a priority to developers. Not to mention the difficulty of reading and writing privacy related policies. In fact , according to Zibran et al. [80], API documentation is among the top factors that negatively affects API usability. Mobile hybrid platforms are no exception. Thus, solutions to solve these problems have been directed towards following a human-centric approach in API design not only to improve usability, but also to promote security and privacy earlier in the design process. Myers et al. [51] provide insights into understanding how different API stakeholders (API designers , API users ,and API consumers) have

different needs- contradictory at some point - which effects the way the API is designed and built. They stress the importance of adopting a human-centric approach to promote usability. Having the API user in consideration when designing the API and investing on tooling support may positively affect API usability. Green and Smith [34] have also supported the argument of creating developer-friendly and developer centric approaches to promote usable security. They have identified ten principles for creating usable and secure APIs. Examples include making the API easy to use, even without documentation and ensure that defaults are safe and never ambiguous.

Along the same lines, implementing secure code has been receiving demonstrable attention recently. Practices followed by developers, such as lapses of attention to security and the lack of application security knowledge in general, are the main reasons motivating this research toward providing more support to developers in order to produce secure code. Jing et al. [72] propose a tool named ASIDE (Application Security IDE), an Eclipse plugin for Java. The plugin goal is to help programmers prevent errors by providing interactive code annotation and refactoring. The tool evaluation [74] reveals that programmers are only likely to successfully address such non-functional errors during programming if the effort and cognitive burden is sufficiently small. Moreover, programmers need easy ways of understanding and learning about secure programming regardless of their experience level.

The need to provide a detailed context rich interface for system configurations has been already discussed for different technologies. Raja et al. [56] have discussed the effect of the Windows Vista personal firewall basic interface on users' mental models. The UI does not present enough details which may result in users developing

an incorrect mental model of the protection provided by the firewall. They suggest a prototype to support development of a more contextually complete mental model through inclusion of network location and connection information. The study shows that participants produce richer mental models after using the prototype. In their analysis of the causes of software errors in terms of chains of cognitive breakdowns, Ko and Myers [45] indicated that certain type of errors are due to attentional issues such as forgetting or a lack of vigilance.

3.4 Security-By-Contract ($\mathbf{Sx}\mathbf{C}$) on Mobile Code

In 2007 [31], the term Security-By-Contract ($\mathbf{Sx}\mathbf{C}$) was proposed as a mechanism to secure mobile apps through using digital signature not only to prove trusted sources but also to bind together the code with a *contract*. Contract is a formal specification on how an app should behave. Every app carries its own contract which can be specified by the developers, the distributor (e.g., marketplace), third-party certification authority, or by the user[30]. The term is interchangeably used with *policy* which is also refers to is a formal complete specification of the acceptable behavior of applications to be executed on the platform for what concerns relevant security actions such as API calls and OS calls. The contract/policy is specied as a list of disjoint rules with a specific grammar. The contract/policy can capture application run-time behavior by either learning some properties of the code or froming an intermediate-level language (e.g., byte-code) or directly from the binary code. Alternatively, a contract can be defined by exploiting the information learnt from the applications executions, e.g. by monitoring some executions of the program to extract its behavior. This overlaps

with the approach we are using to secure hybrid apps by monitoring app behavior to generate a policy to be enforced later. However the approaches are different in several aspects. First, our approach captures the behavioral properties of the app based on the monitored execution. This approach is based on specifying rules (by developer or user) to govern app behavior in a context. Our approach is captured in the development/testing stage while this approach can be enforced in all three stages (development, deployment and execution). Most of the literature have discussed this notion under Java and .Net frameworks depending basically on binary code an app rewriting. Our approach domain is different and the implementation details.

CHAPTER 4: HYBRID APPS STATUS QUO: SECURITY & STATISTICS

In this chapter we reflect and analyze the current situation of hybrid apps in terms of the configurations/ development practices and trends and we discuss how these can affect apps security and privacy. We first present the results of the market analysis of real-world hybrid apps that was conducted on 2013-2014. Our main goal of this study is to analyze current configurations schemes and discover common trends of hybrid apps development at the time. Due to the fact that the library is fastly changing and its configuration and security measures are evolving, we conducted another large scale market analysis in 2017. We have addressed new configurations items and their usages. We have also highlighted actual developers' usage patterns of the security mechanism provided by the library.

Then we present threat models that apply to hybrid apps by discussing how apps are compromised and how this can maliciously affect the user and the device. We also demonstrate how “bad” configurations can play a role in enabling the malicious code to be triggered and amplifying its damage.

4.1 Market Analysis - 2014

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.

4.1.1 Data Collection

The PhoneGap website [38] hosts a repository linking to different app markets for apps developed using the PhoneGap framework. We downloaded all free PhoneGap Apps (662 apps) that were built over Android from Google Play [33] and were showcased by PhoneGap in 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 the `www` folder. We built a C++ tool to parse and extract information from the retrieved files, which includes the app permissions, plugin declarations, plugin usages and other configuration details.

4.1.2 Results

4.1.2.1 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 most Hybrid platforms consider using *SPA* a good practice for many reasons related to performance and simplicity. However, Figure 14 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. 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. 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

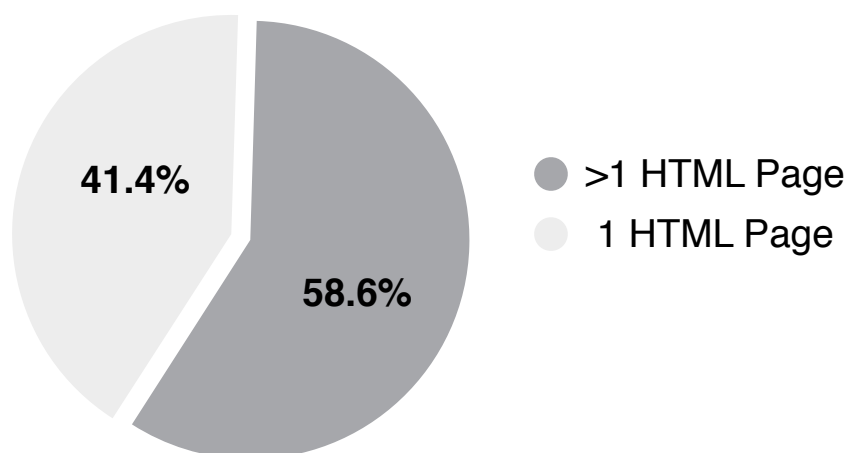


Figure 14: HTML File App Count

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 $sim(x, y) = \frac{x \cdot y}{\|x\| * \|y\|}$ to compute the similarity between the pages in the same app. Figure 15 shows the average page similarity distribution for scanned apps, whereas the 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.

4.1.2.2 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 16 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

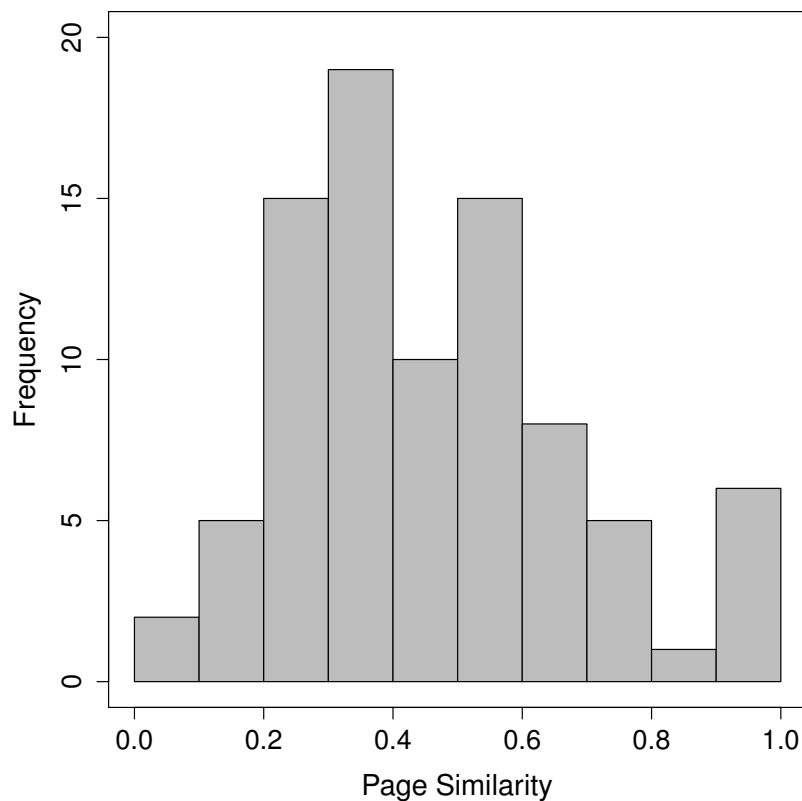


Figure 15: PhoneGap dataset page similarity distribution

GeoBroker plugin and only 8% of these apps actually use this plugin. This over declaration can be easily exploited by maliciously loaded scripts.

4.1.2.3 Access Origin Usage Patterns.

As discussed earlier, domain whitelisting is the current security model adopted by PhoneGap to control access to outside domains and subdomains. By default access is open to any domain `<access origin="*" />` which means that an external server can communicate with this app. 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 the apps in our dataset, and we cate-

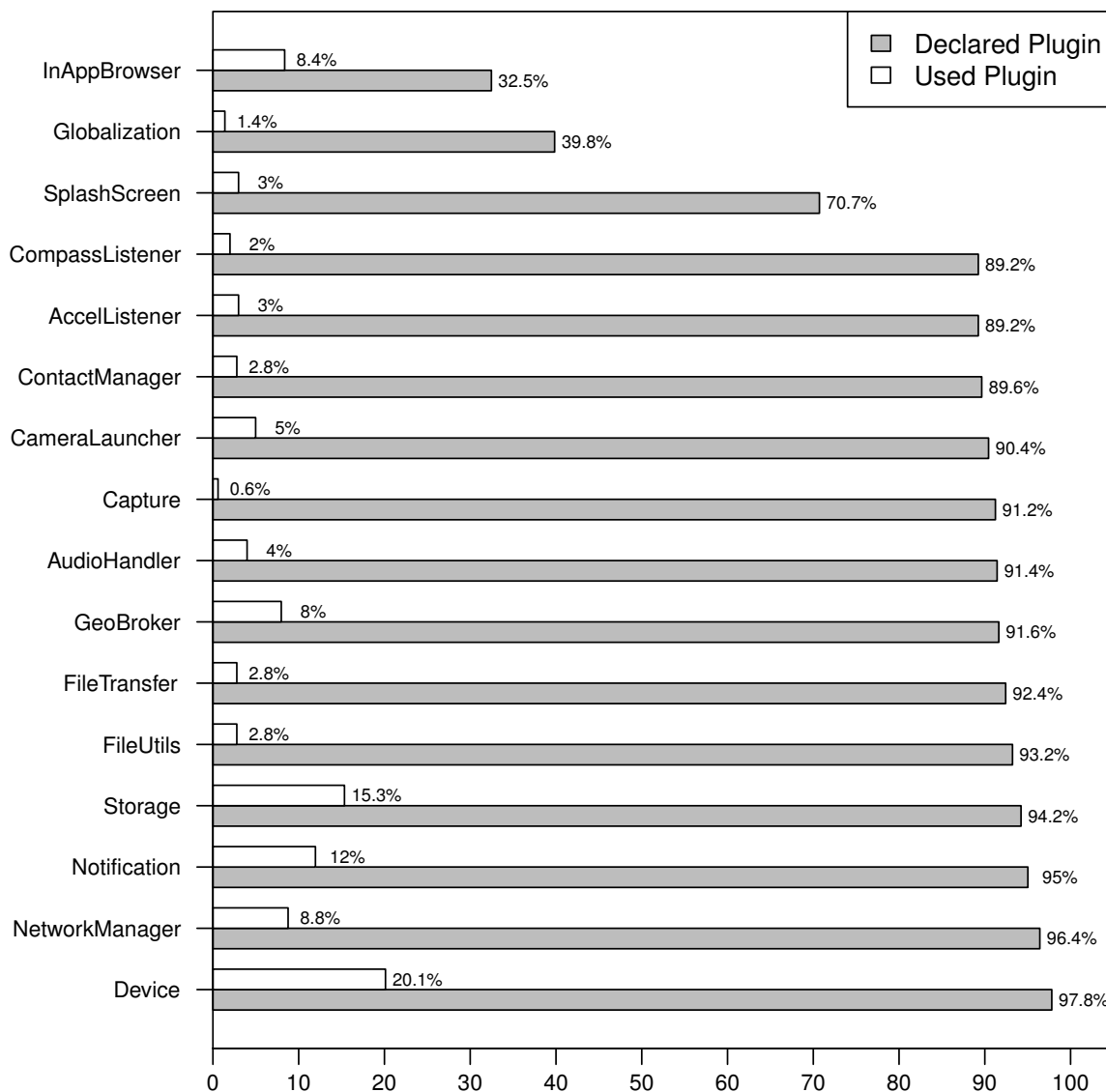


Figure 16: Plugin declaration vs plugin usage

gorized 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 17 shows the different access origin category combinations 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.

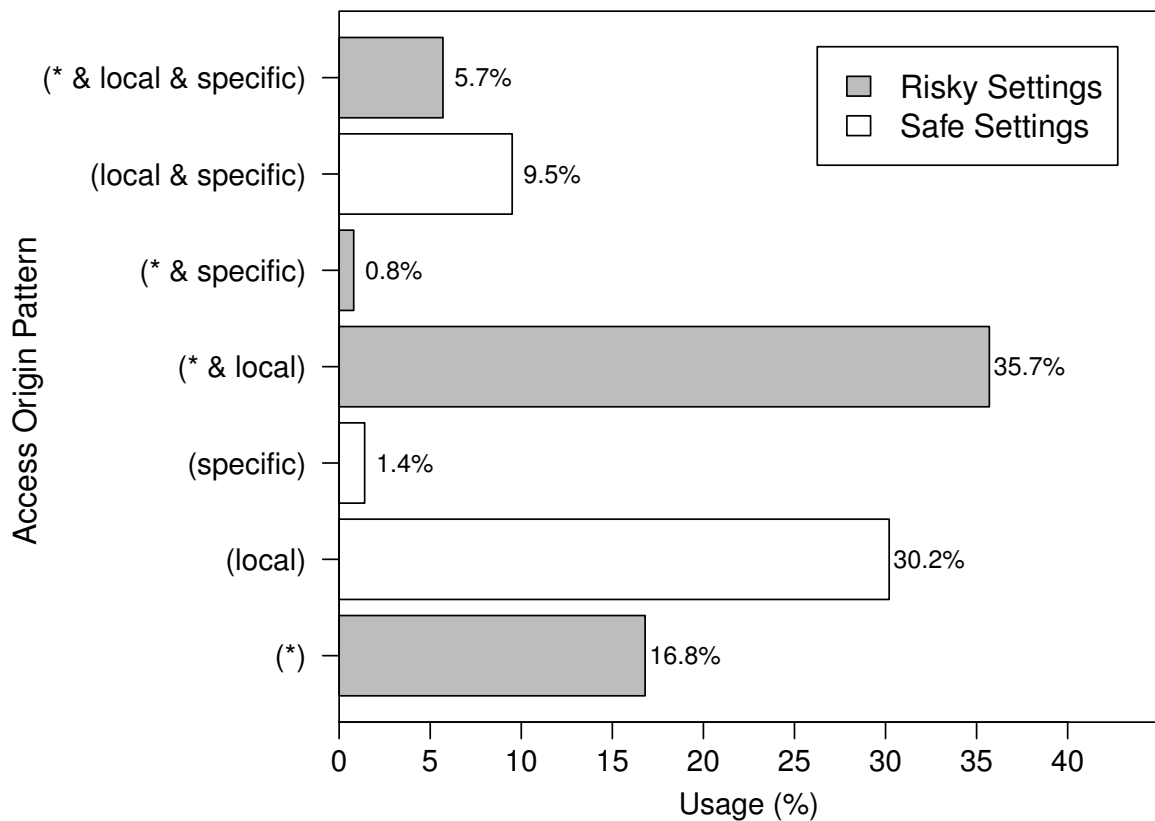


Figure 17: Access Origin usage distribution

4.2 Market Analysis - 2017

In this subsection we present a large-scale study that is centered on mobile hybrid apps configurations and permissions usage patterns. During the past five years, the library have evolved in different aspects. The configuration model has evolved to have

more fine-grained configuration items to capture different aspects of the app behavior. It also added new security tools such as using CSP in the default starting page. We want to (1) discover how developers are configuring the apps; (2) how effective these additions by the middleware; and (3) if there exist any pattern of using these new configurations. We find that while the platform is adding more security features, there is a demonstrable misconfiguration trend. The result of analyzing a set of 2111 hybrid apps uncovered several alarming observations. We have found that 80% of the apps are vulnerable to injection attacks because of an absence or a poor usage of the security model provided by the platform. We also detect a trend of keeping risky default configuration settings which results in having over-privileged apps that may expose device APIs to malicious code. On the system side, we realize that most of the apps have access to the platform's INTERNET and GEOLOCATION permissions. Google messaging is also recognized as the most widely used third-party service. In addition, we detect a suspicious set of domains including spying, payment, Adware, and military that are white-listed.

4.2.1 Data Collection

We compiled a set of cordova based package names from different resources including previous research work² in addition to package names manually extracted from the platform website. To ensure that the apps are still active in Google Play market, we have implemented and used an automated tool to search for the app name in Google Play, install the apk on a mobile device, then copy the apk file from the

²We obtained the data set used in the paper [41]

device to a computer. We started with a list of 20000 names and ended up with a 2111 apps. The selection criteria was based on the following: - The app exists in Google play

- The app installs properly on a mobile device
- The app package (apk file) is not corrupted and was transferred successfully from the mobile device to another computer.

4.2.2 Results

4.2.2.1 Starting Page<src>

A hybrid app consists of one or more pages. Pages can be hosted locally inside the app folder or remotely. We have found that most of the apps (98%) start from a local page. The rest of the apps (2%) have the following settings :

- 17 apps are set to start from a remote page that uses **http** connection
- 3 apps are set to start from a remote page that uses **https** connection

4.2.2.2 Network Resources Access<access>

This is a key setting as it indicates the white-listed domains that can download resources into the app, including Javascript code. The code from a white-listed domain has the privilege to access device APIs and execute on the device just like the local code. The developer may use one or more rules to white-list the domains. The library adds a default rule of a “*” which restricts no domain. Developers may remove it and specify domains according to the app needs.

We first report how many rules are found for this configuration item. Results of

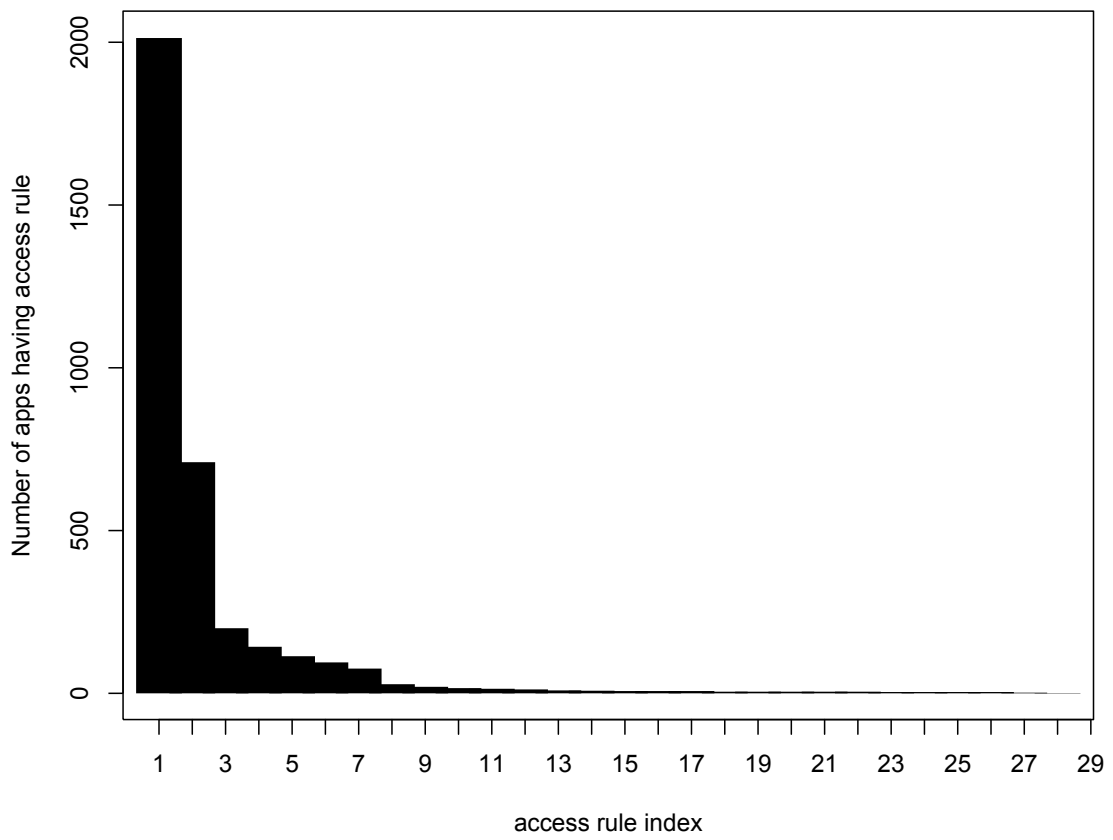


Figure 18: Access Rules Usage

scanning the data set indicate that most apps (96%) use 1 to 5 rules (see Figure. 18). A maximum number of rules per app reached up to 29 access rules.

In terms of rules' values, we categorize them into system provided settings that are suggested by the platform and custom settings that developers set depending on the app specific access needs. Figure. 19 demonstrates the percentages of apps using default values provided by the platform. The value “*” has the highest percentage of occurrences as **67%** of the apps include this value. This means that almost **1342** apps are configured to accept downloads from *any* server. Localhost value which is found in 37% indicates that the app can accept accesses from local files. This low percentage can be attributed to the fact that setting the access to “*” implies that it can download from itself so developers may not need to add it. Values such as "tel:/*" and "http:/*/*" are found because in versions before 5.0.0, the items `allow-navigation` and `allow-inent` were not added yet. Hence, `<access>` was the only configuration item related to domain white-listing regarding network resource access, intents, and navigation. Other values that are related to intents and navigation are found in less than 5%.

Developers may also use specific domains which is a recommended practice by the library. Yet, as it is demonstrated in Figure 20, custom URLs are found in less than 5% of the apps. We categorize URLs based on their purpose. Social media URLs such as `Facebook`, `Twitter`, and `Youtube` are found in 3.4% of the apps. Next, Google APIs, such as “*.googleapis.com”, “http://google-analytics.com”, and “https://play.google.com/store/apps/*”, are found in 2.4%. We have also found that almost 25 apps (1.1%) white-list the `Gstatic` domain which can be explained be-

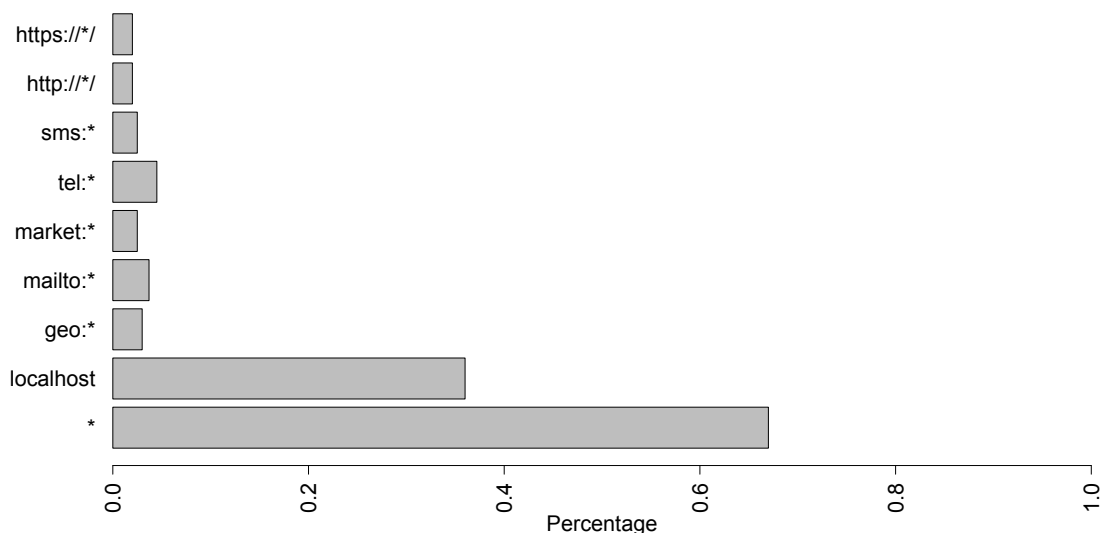


Figure 19: Network Resource Access: Platform Provided Settings

cause it is needed for TalkBack function [5]. On the other hand, this URL is considered by experts [10][6] a malicious URL that works in a way very similar to an Adware. Others [12] indicate that this domain is for tracking page loads on certain sites. Moreover, we have identified several URLs (“https://*.netspend.com”, “https://*.mycontrolcard.com”, “https://*.wuprepaid.com”, “https://*.acebusinessselectcard.com”, “https://*.paypal-prepaid.com”) related to online payment/banking in 0.8%. Additionally, spying URLs (“https://*.spykemobile.net”, “https://*.iesnare.com”) are found in 0.5%. Finally, we identify military and government domains in 0.2% of the apps.

4.2.2.3 Domain White-listing for Intents <allow-intent>

Starting from version 5.0.0, this configuration item is added to control the app inter-process communication between the app itself and other apps and also what data can be passed. We have found that only 15% of the apps contained settings

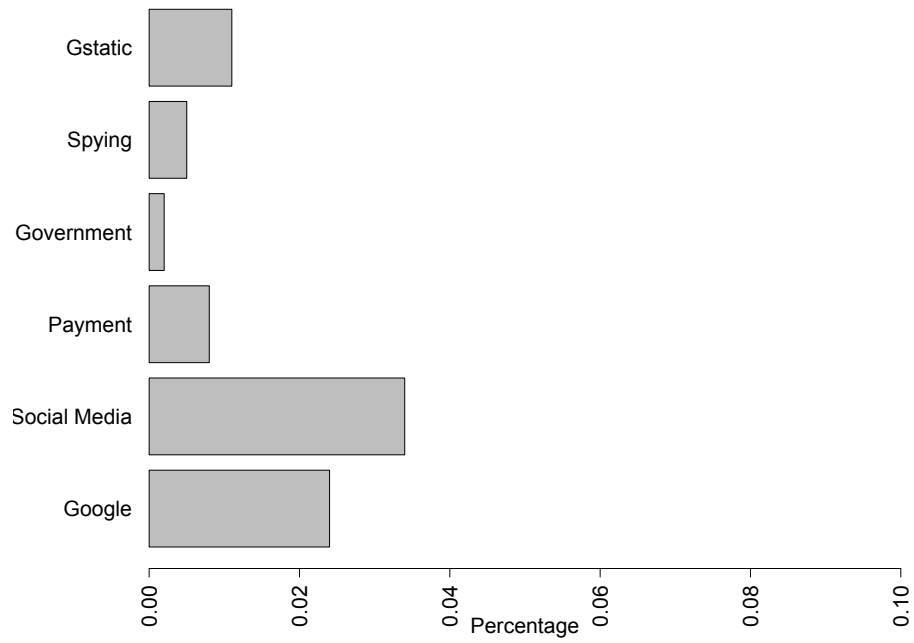


Figure 20: Network Resource Access: Custom Settings

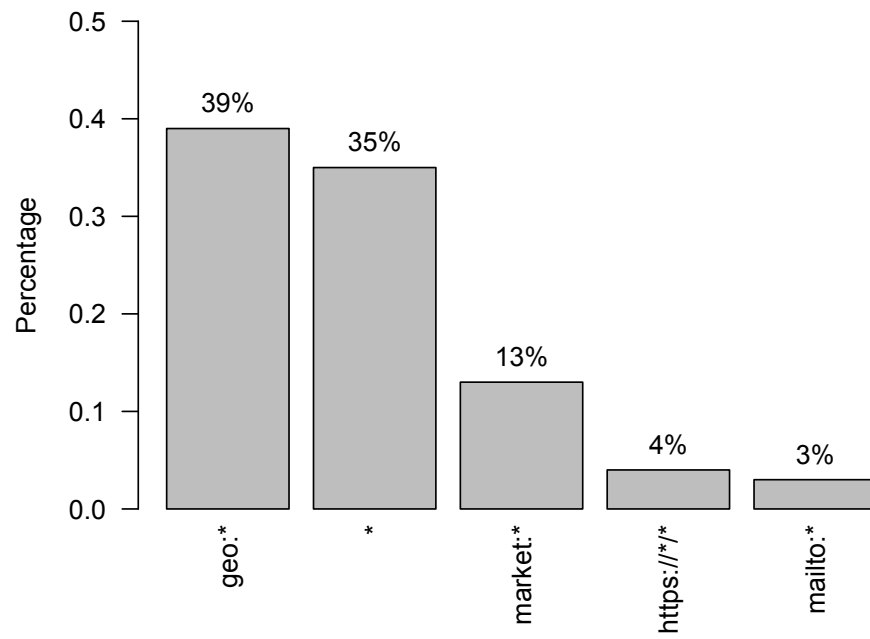


Figure 21: Intents White-list Settings

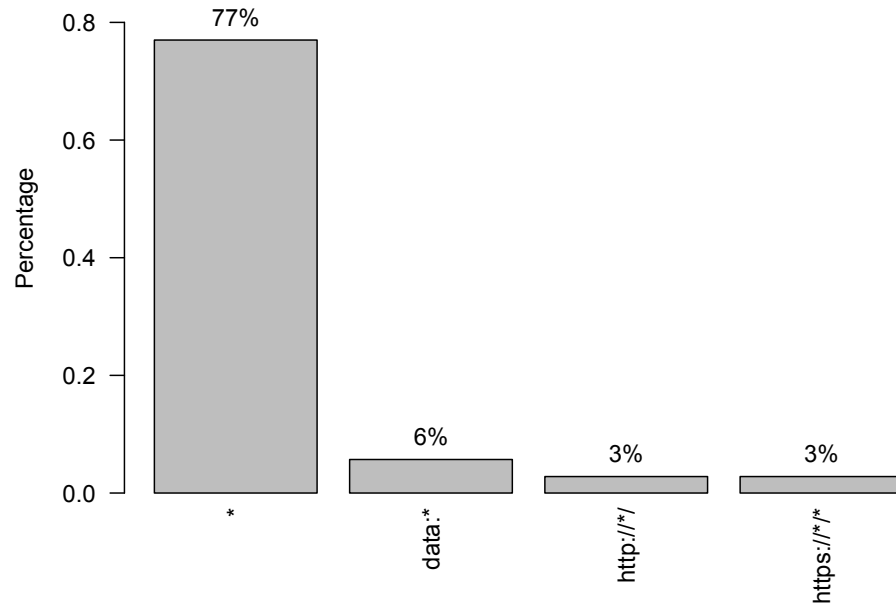


Figure 22: Navigation White-list Settings

related to intents' white-listing. For that set, we demonstrate the configuration values distribution in Figure. 21. We can see that among those apps, 39% white-list “geo:*” which offers maps service, followed by 35% using “*” which allows interaction with all apps. The intent that white-lists Google play (“market:”) is found in 13%. Then, the rest of the values are found in less than 5%.

4.2.2.4 URL Navigation <allow-navigation>

We have found that only 10% of the apps contain navigation white-listing settings. In that set we scan the settings and demonstrate the distribution in Figure. 22. For this configuration item, there are no default configurations. This means that any added configuration is added by the developer. The value “*” is found in 77% of the

apps which allows navigation to any protocol/ domain. This might explain the low percentages of all other values ,such as “http://*”, “https://*” and “data://*”, since the value “*” includes them all.

4.2.2.5 Plugins Usage <feature/plugin>

We scan plugin configuration tags and divide them into core and custom. It is important to highlight that plugin declaration in the configurations does not necessarily mean their usage. The plugin tag may exist because it is added by default, which may explain the high percentages of default core plugins usage as depicted in Figure. 23. This also explains why 88% of the apps use core plugins.

On the other side, 80% of the apps use custom plugins in addition to the core ones. We scan and classify custom plugins based on their purpose. Figure. 24 shows custom plugins usage percentages. Plugins related to accessing device features are found in 79% of the apps. Examples of device related plugins are bar-code scanners, location based, and SMS plugins. Application APIs mostly related to web browsing are found in 26%. Video related APIs such as video players are found in 25% of the apps. We also identify several encryption APIs (Crypto, i4crypt ,and cryptUtil) in 7% of the apps. Application user interface interaction APIs such as push notifications and progress dialogues are found in 6%. Same percentage for social media APIs such as Facebook, Twitter, and social sharing. Advertisement APIs and screen image APIs are detected in almost the same percentage (1%).

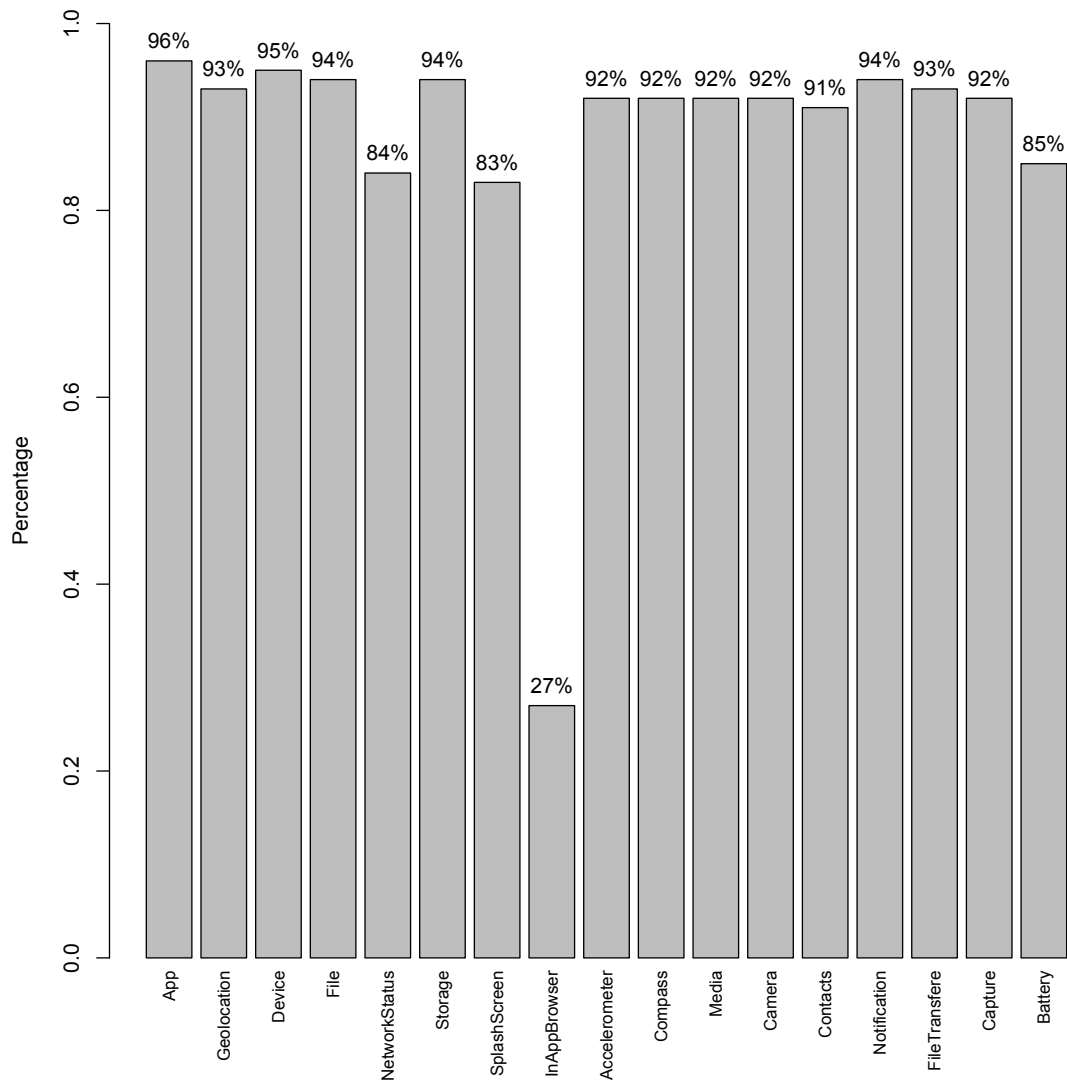


Figure 23: Core Plugins APIs

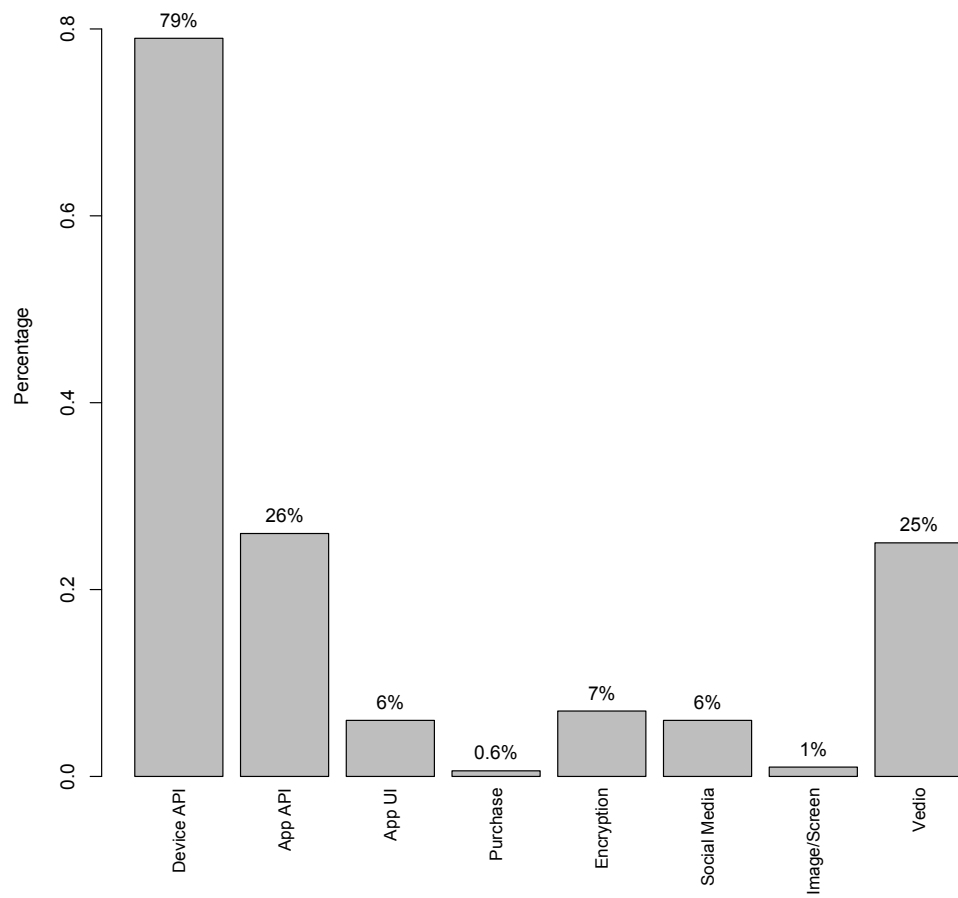


Figure 24: Custom Plugins APIs

4.2.2.6 Native Permissions

In a hybrid app, using a plugin may require one or more platform permissions on the native side. For instance, using the CAMERA plugin on an Android app requires the app to use the permission “android.permission.CAMERA”. Including a plugin in the configuration without using the proper corresponding native permissions shall void any call to that plugin because the native platform won’t be able to fulfill the request. Permissions in Android is a security mechanism by which the platform protects access to sensitive device data and sensors. Developers need to include proper permissions to use certain functionalities. In addition to platform defined permissions, a developer may define her own custom permissions to control access to a component from other apps .

Android permissions are divided into several protection levels. The two most common levels are normal and dangerous permissions. Normal permissions are used to grant access to data or resources outside the app’s sandbox, but where there’s very little risk to the user’s privacy or the operation of other apps. However, dangerous permissions grant access to data or resources that involve the user’s private information, or could potentially affect the user’s stored data or the operation of other apps [1].

We are particularly interested to scan the permissions used to see if there exists a particular set of permissions that are often used. We are also interested to check if these native permissions are the required permissions for the default plugins. To extract native permissions we scan the file AndroidManifest.xml file. We have found

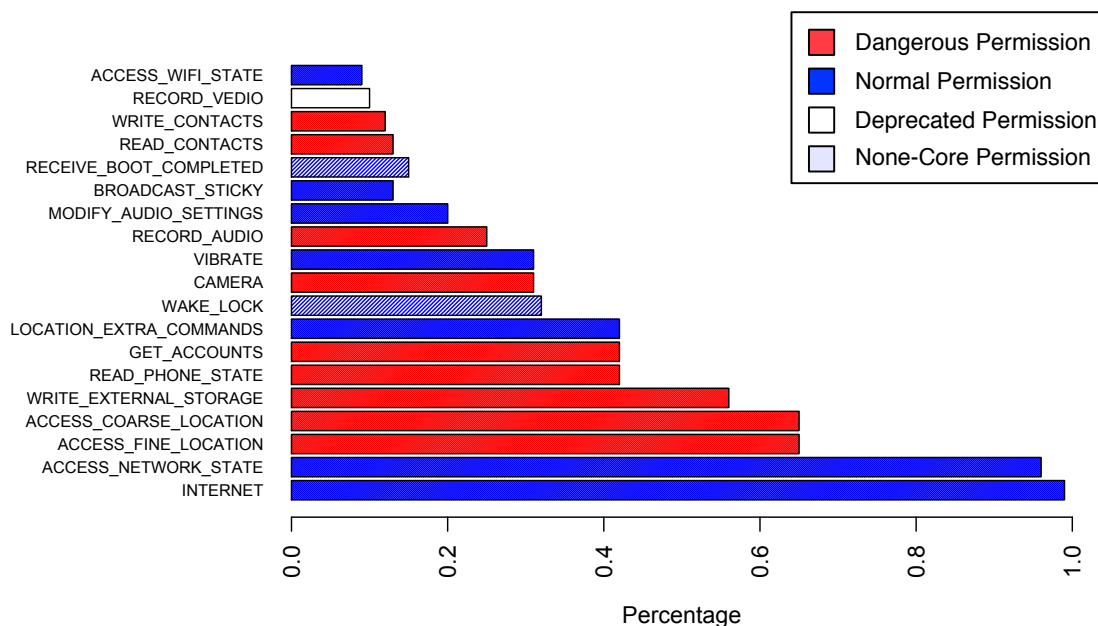


Figure 25: Platform Native Permissions Usage

that 90% of the permissions used are standard Android permissions. The 10% are developer's customized permissions to use third party services such as google cloud messaging and In-app billing.

For the standard permissions, we show in Figure. 25 the top permissions that are color coded based on the protection level. Blue bars represent normal permissions while red ones represent dangerous permissions. Solid bars are for permissions needed for core plugins, while striped ones are permissions needed for other purposes. The permission to use INTERNET is used in almost all the apps. Then comes a set of dangerous permissions. Examples include those related to accessing location such as ACCESS_FINE.LOCATION and ACCESS_COARSE_LOCTION. Also the WRITE_EXTERNAL.STORAGE permission that is needed to save data in the de-

vice, the permission `READ_PHONE_STATE` that allows read only access to phone state, including the phone number of the device, current cellular network information, the status of any ongoing calls, and a list of any Phone Accounts registered on the device.

While most permissions are those needed to use the core cordova plugins, we found that there are two permissions; `WAKE_LOCK` (32%) and `RECEIVE_BOOT_COMPLETED` (15%) that are not needed for the core plugins, yet they are among the top used permissions. Both are normal permissions. The first one allows using `PowerManager WakeLocks` to keep the processor from sleeping or the screen from dimming. The second one allows an application to receive the broadcast after the system finishes booting [2].

We also found that the permission `RECORD_VEDIO` that exist in 10% of the apps does not actually exist on the Android permission list. Its' existence is probably due to being a default permission added by the library in previous versions.

Regarding custom permissions, we have found that the most used permission is the “`com.google.android.c2dm.permission.RECEIVE`” which is used in 36%. This permission is needed to use Google’s cloud messaging service. Next permission is the in-app billing permission “`com.android.vending.BILLING`” which is used in 2%.

It is observed that the permissions that are added by default are found in high percentages (average of 91%). This also supports the reflection that developers tend not to change the default settings. This is also an indication that these granted permissions exist not because they are necessarily needed, rather because they are simply added by default. This also indicates an issue of permission misuse and violating the

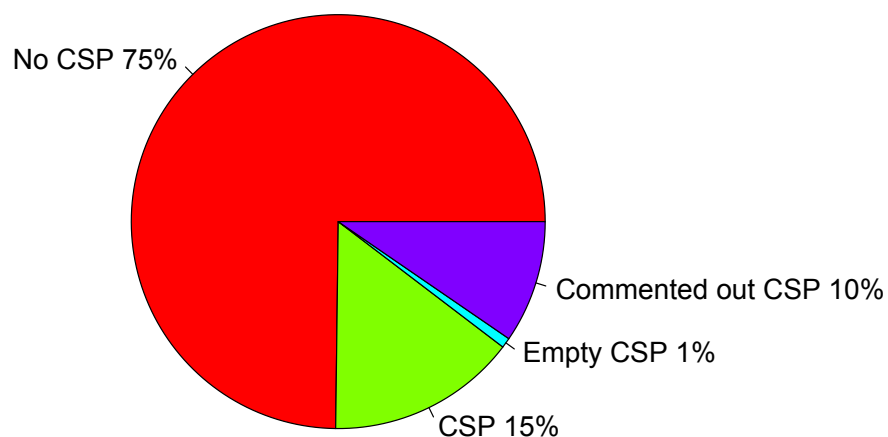


Figure 26: Content Security Policy Usage

Least-Privilege principle.

4.2.2.7 Content Security Policy

Unlike previous configuration items, CSP is not set in the `config.xml` file. It is set in the html pages. The purpose of setting CSP per page is to control content download. We scan the `index` page and extract CSP meta tags- if any. Results depicted in Figure. 26 show that the majority (75%) did not contain any CSP. Moreover, 10% of the apps contained a commented out CSP. These are probably the default generated CSP tags but for some reason, developers decided to comment them out. We also identified empty CSP rules in 1% of the apps. This leaves only 15% of the apps having an active CSP. This means that **85%** of the apps have no effective CSP enforcement. Thus, no control of content download on a page level which means that these apps are vulnerable to injection attacks. To inspect the way policies are

Table 5: Policies Break Down

| Policy Directive | Policies Values | | | | |
|--------------------------|-----------------|---------------|-------------|-----------|------|
| | * | unsafe-inline | unsafe-eval | self only | URLs |
| <code>default-src</code> | 79% | 65% | 70% | 2% | 2% |
| <code>script-src</code> | 63% | 59% | 1% | 11% | 0% |
| <code>style-src</code> | 5% | 27% | 0% | 24% | 0% |
| <code>img-src</code> | 6% | 1% | 0% | 2% | 1% |
| <code>media-src</code> | 9% | 0% | 0% | 0% | 0% |
| <code>frame-src</code> | 1% | 0% | 0% | 0% | 0% |

specified in the 15% of the apps (178 policy), we break down the policies and report results in Table 5. Each percentage is the number of occurrences of the policy found for the corresponding policy directive divided by the total number of policies (178). For instance, 79% of the policies allow content download from any domain. Considering the way policies are set, we argue that developers are using liberal values in setting their policies. Having * as the most commonly used value rather than specifying URLs is an example. Moreover, allowing unsafe-inline and unsafe-eval enables malicious code execution. Strict policies such as using 'self' or specifying URLs are the least used. A combination of * as a source of script download, allowing unsafe-inline or unsafe-eval voids the whole purpose of having CSP. This combination is found in 62% of the policies.

4.3 Cordova Common Vulnerabilities (CVEs)

The Cordova community has been evolving rapidly. CVE Details [27] have dedicated a page for Cordova-Specific common vulnerabilities and exposures[28]. Table 6 shows the CVEs discovered up to the time of the writing. Most of the Cordova vulnerabilities are Bypassing either URL or device-resource access.

Table 6: Cordova Security Vulnerabilities[28]

| CVE ID | Type | Publish Date | Version | Platform |
|---------------|-------------------------|--------------|---------|----------------------|
| CVE-2015-8320 | Bridge Hijacking attack | 2015-11-23 | 5.0 | Android |
| CVE-2015-5256 | Whitelist Bypass | 2015-11-23 | 4.3 | Android |
| CVE-2015-5208 | Whitelist Bypass | 2016-05-09 | 4.3 | iOS |
| CVE-2015-5207 | Whitelist Bypass | 2016-05-09 | 4.3 | iOS |
| CVE-2014-3502 | Send data to apps | 2014-11-15 | 7.5 | All |
| CVE-2014-3501 | Whitelist Bypass | 2014-11-15 | 4.3 | Android |
| CVE-2014-3500 | Change the start page | 2014-11-15 | 4.3 | Android |
| CVE-2014-1884 | Device-Resource Bypass | 2014-03-02 | 6.4 | Windows Phone 7&8 |
| CVE-2014-1882 | Access Bridge | 2014-03-02 | 7.5 | All |
| CVE-2014-1881 | Device-ResourceBypass | 2014-03-02 | 7.5 | All |
| CVE-2012-6637 | Whitelist Bypass | 2014-03-03 | 7.5 | All |

CVE-2015-8320 Apache Cordova-Android before 3.7.0 improperly generates random values for BridgeSecret data, which makes it easier for attackers to conduct bridge hijacking attacks by predicting a value.

CVE-2015-5256 Apache Cordova-Android before 4.1.0, when an application relies on a remote server, improperly implements a JavaScript whitelist protection mechanism, which allows attackers to bypass intended access restrictions via a crafted URI.

CVE-2015-5208 Apache Cordova iOS before 4.0.0 allows remote attackers to execute arbitrary plugins via a link.

CVE-2015-5207 Apache Cordova iOS before 4.0.0 might allow attackers to bypass a URL whitelist protection mechanism in an app and load arbitrary resources by leveraging unspecified methods.

CVE-2014-3502 Apache Cordova Android before 3.5.1 allows remote attackers to open and send data to arbitrary applications via a URL with a crafted URI scheme for an Android intent.

CVE-2014-3501 Apache Cordova Android before 3.5.1 allows remote attackers to bypass the HTTP whitelist and connect to arbitrary servers by using JavaScript to open WebSocket connections through WebView.

CVE-2014-3500 Apache Cordova Android before 3.5.1 allows remote attackers to change the start page via a crafted intent URL.

CVE-2014-1884 Apache Cordova 3.3.0 and earlier and Adobe PhoneGap 2.9.0 and earlier on Windows Phone 7 and 8 do not properly restrict navigation events, which allows remote attackers to bypass intended device-resource restrictions via content that is accessed (1) in an IFRAME element or (2) with the XMLHttpRequest method by a crafted application.

CVE-2014-1882 Apache Cordova 3.3.0 and earlier and Adobe PhoneGap 2.9.0 and earlier allow remote attackers to bypass intended device-resource restrictions of an event-based bridge via a crafted library clone that leverages IFRAME script execution and directly accesses bridge JavaScript objects, as demonstrated by certain cordova.require calls.

CVE-2014-1881 Apache Cordova 3.3.0 and earlier and Adobe PhoneGap 2.9.0 and earlier allow remote attackers to bypass intended device-resource restrictions of an event-based bridge via a crafted library clone that leverages IFRAME script execution and waits a certain amount of time for an OnJsPrompt handler return value as an alternative to correct synchronization.

CVE-2012-6637 Apache Cordova 3.3.0 and earlier and Adobe PhoneGap 2.9.0 and earlier do not anchor the end of domain-name regular expressions, which allows remote attackers to bypass a whitelist protection mechanism via a domain name that

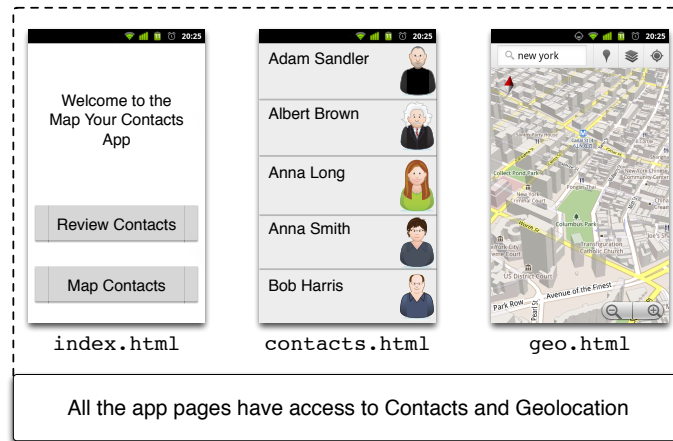
contains an acceptable name as an initial substring.

CHAPTER 5: SECURING HYBRID APPS THROUGH THE APP CONFIGURATIONS

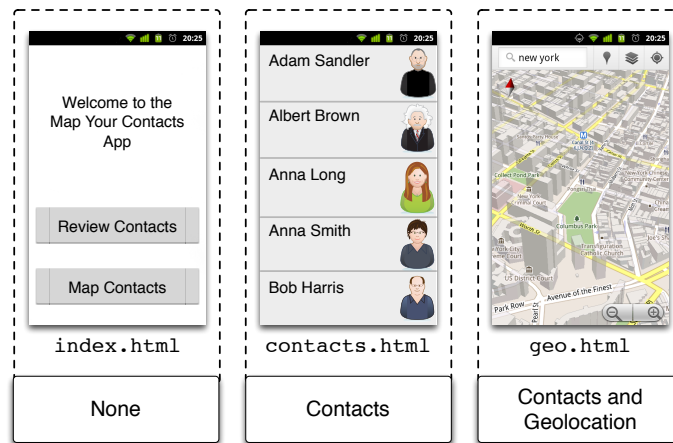
In the previous section we have demonstrated how the current configuration scheme of the library suffers security limitations that can compromise apps' security, especially if the developer does not change default configurations values. In this section, we demonstrate two methodologies we have developed to have more aligned and coarse-grained configuration model. Both approaches propose a finer-grained configuration scheme and suggest more aligned setting values.

5.1 Page Level Configuration Model

The Cordova access policy allows the developer to specify the list of whitelisted domains and the set of plugins (features) to be included in the app. This model specifies 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 [60] since extra permissions and access to plugins is granted to loaded pages that do not require such accesses. Figure 27(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



(a) Current Cordova plugin access Model



(b) Proposed Least Privilege Approach

Figure 27: Example Multi-Page App and Access Models

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 policies by slightly modifying the Cordova library. Figure 27 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 some functions in the contacts plugin and the `geo.html` page is granted access to some functions in the contacts and geolocation plugins. In order to implement the proposed framework in the context of a cordova-based app, we let the app go through two stages; “build” which means that the access policy table is being composed; this phase is meant to be while the app is either under the development or in the testing phase, “enforce” which means any access to a plugin should conform with the policy table; this phase is meant to be when the app is to be released to the market. To do this we add one element `<policy />` to the `config.xml`, see figure 28. To design the model, these questions should be answered:

1. How to build a plugin access policy?
2. Where to store the proposed policy rules?
3. Where should the reference monitor be implemented?

To address the first and the second challenge, we monitor app plugin access behavior


```

<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>

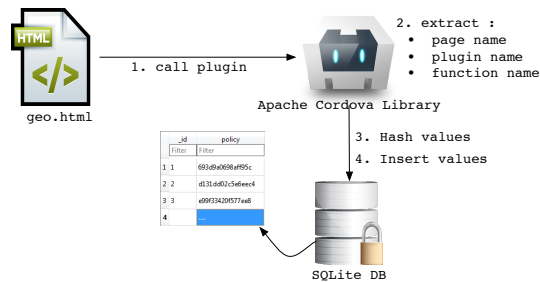
```

Figure 28: Policy stage in config file

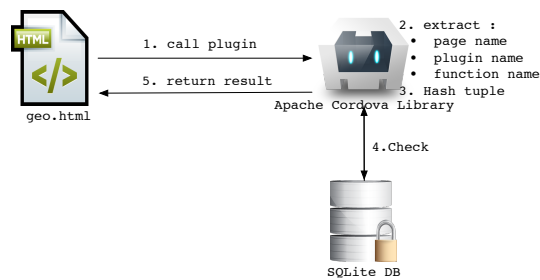
in terms of page, plugin and action requested while the app is in the “build” stage. As demonstrated in figure 29(a), in this stage, any plugin access done during the development and the testing phase is considered a valid plugin rule, as a result, it is automatically recorded, hashed and added to the database as an access policy record. This saves the developer specifying any rules manually, since the assumption here, is that through app development and testing most if not all possible plugin calls are covered as a normal testing procedure. These rules are hashed and then saved in a SQLite data base. Once the app is released to the market -shipped with the database that has the access policies-, the enforce flag is set which means stop writing on the database and enable authenticating every plugin access request against the saved policies, see figure 29(b). The change is transparent to the developer since it doesn't

require the developer to do any extra step regarding setting or enforcing policies.

The access control model follows the closed world assumption, where if access is not



(a) Build Stage



(b) Enforce Stage

Figure 29: Build/Enforce Policy

explicitly specified then it is assumed it is not granted.

We updated the `PluginManager` plugin mapping logic to accommodate the plugin access rule 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 gen-

erate a `PluginResult` that includes an illegal access permission message, which will throw an exception in the JavaScript.

```
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 30: Proposed PluginManager Rule Check

Figure 30 shows the reference monitor code inserted in the `PluginManager` execute method.

5.2 Behavior-Based Configuration Model

Configuring any application requires a profound understanding of its behavior. Static and dynamic code analysis approaches combined provide a comprehensive methodology of modeling not only application behavior but also other details such as data flow and control flows. Static analysis of cordova-based apps has been done in previous research [19]. While this approach may provide an accurate call graph with respect to static behavior in terms of cross-language calls, dynamic app features may not be present in this analysis. Mobile Hybrid Apps are dynamic by nature. Changing app logic, especially through remote Javascript calls is reasonable. Moreover, API calls using obfuscated code may not be captured as well. Hence, we present a dynamic behavior modeling.

Our proposed approach captures *(state,plugin)* access rules by monitoring the app

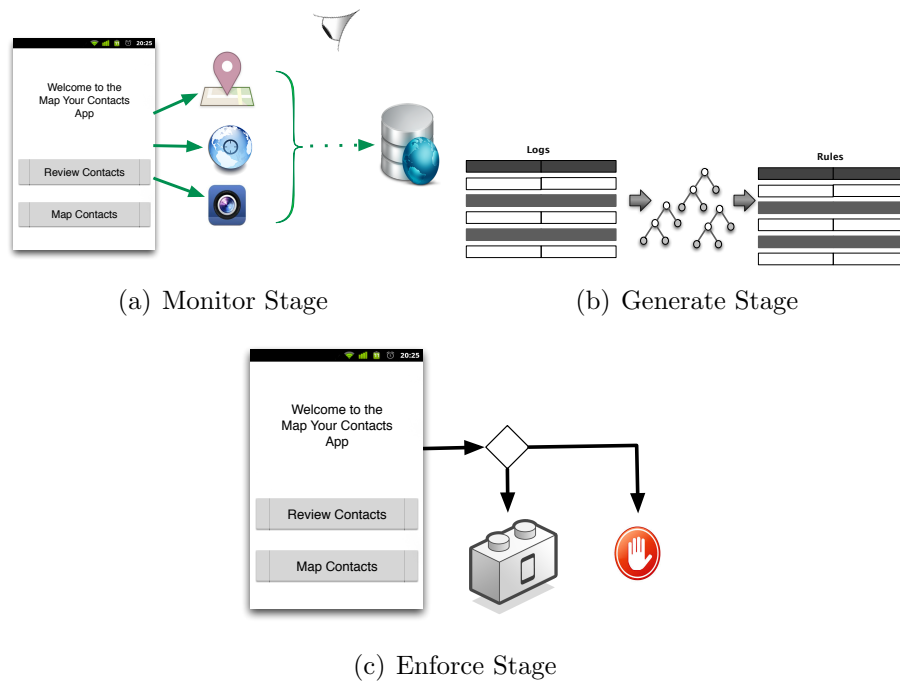


Figure 31: Three-Stage Behavior Policy

behavior while running in a controlled environment. It also captures the *states transitions* sequence and then generates a policy to control app states transitions. To implement these changes we have changed the internal implementation of the library. Code changes mainly applies to three classes: `PluginManager`, `CordovaWebView`, and `CordovaActivity`. The addition of code does not exceed 300 LOC. We have published the modified version of the library on Github ³. Our ultimate goal is to control malicious code's impact on the device and the app using more aligned and fine-grained configurations. We have chosen to monitor behavior in the stage of testing for two main reasons:

- We can control code coverage by using extensive testing scenarios that cover all API calls. Recommended App Testing practices require testers to go through

³<https://github.com/aaljarra/ModifiedCordova.git>

all functional scenarios of the app, covering every possible functionality needed to be implemented. Thus, monitoring app behavior at this stage provides rich information about how the app should perform.

- Using an existing stage to monitor the app behavior comes incongruent with our goal of keeping the process as seamless as possible to the developer. Also, to support addressing potential vulnerabilities as early as the development and testing stage.

Our approach is focused on protecting the app against attacks that change the app behavior, aiming to abuse the library implementation through accessing device features or tampering app behavior. If this malicious behavior is detected, then the code will be prevented from execution by setting configuration that defines app healthy behavior.

According to the attack examples mentioned earlier in subsection 2.4, plugins access and app behavior are listed to be potential targets; thus, we have modeled app behavior in two directions, plugin access and state transition as it is explained in the next two sections.

5.2.1 Plugin Access Policy

Our approach enables capturing app behavior and then enforcing access rules elicited from the behavior of the app. The approach is a three-stage solution. The app should go through the *Monitor* stage, *Generate* stage and then *Enforce* stage as shown in Figure. 31. We extend the `config.xml` file to enable the management of tracking these stages by including a “stage” element. The Cordova library contains

a function called `exec`, which serves as a central hub for plugin access. No plugin can be accessed without being passed to this function. Hence, we add our reference monitor logic in the `exec` function.

Algorithm 1 Plugin `exec()` Algorithm

```

1: procedure EXEC()
2:   currentStage = extract stage from config.xml
3:   if currentStage is "monitor" then
4:     execute plugin
5:     call monitor(plugin name, current page state)
6:   else satisfyRules = call enforce(plugin,current page state)
7:     if satisfyRules = true then
8:       execute plugin
9:     else
10:      cancel plugin call
11:      notify user
12:    end if
13:  end if
14: end procedure

```

Hence, we added our reference monitor logic in the `exec` function. Algorithm 1 explains how the *current* stage of the app determines how to deal with a plugin call.

Monitor Stage: Occurs when the app is running in a controlled environment, namely, while in development and testing. Any access request to native plugins is monitored and associated with the app’s current state. Here, “State” refers to the URL parts that include protocol, domain, subdomain path, fragment, and parameters. This stage also captures the operation requested. The current state of the app could end with a page name or a hashbang. Hashbang is anything after “#” tag in the URL which can be either `#/action` or `#action[68]`. For either single or multi page apps, all the URL parts are used to denote a state of the app. As described earlier, each `CordovaActivity` contains a `CordovaWebView` that extends the `WebView` class.

We extract the current state by calling the `WebView` instance's `getUrl()` method. The information extracted are saved as Logs in a database.

Generate Stage This is an intermediate stage that is necessary to generate a behavior-based policy for the app. It is meant to be when the developer is done with development/testing and decides to release the app to the market. This stage is done once, when the app stage in `config.xml` is changed from the *Monitor* to *Enforce*. This change triggers the process of reading logs that were previously saved during the *Monitor* stage to extract a fine-grained policy that represents the app behavior in terms of state-plugin rules. Processing the logs passes through the following steps:

- I Build Call Map: This step reads all plugin access instances and groups them by plugin. As a result, a map of plugin, and a set of URLs are generated. This map (`callMap`) helps construct access rules per plugin in the second step.
- II Build Call Tree: This stage is essential to group all app URLs into states through a syntax tree that represents all captured URLs of a specific plugin. For each plugin a syntax tree (`PluginCallTree`) is generated to represent the states such that it can be traversed later to generate a set of expressions that can be used for enforcement. The pair (`plugin,PluginCallTree`) is saved into another map `callTreeMap` (see algorithm 2).
- III Extract Patterns: This step traverses the `callTreeMap` to extract regular expressions for every plugin. The list of regular expressions will be used to form the access policy used by the third stage `enforce()`.

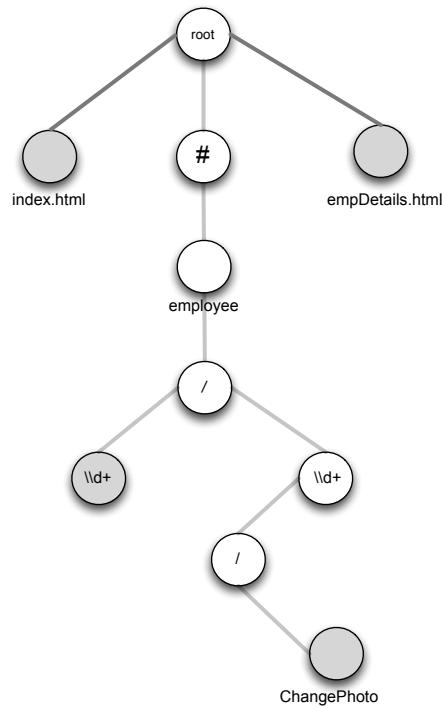


Figure 32: Abstraction of Syntax Tree Representing States

A simple heuristic function is implemented for step II that is responsible for identifying app states. For multi-page apps, URLs can be used as is to serve as identifiers for the app states. In single-page apps (SPA) however, URL fragments are considered. Generally, SPA URLs follow the following scheme:

$$\#[/]action[[/index]^* [/action]^*]^*$$

Where action is an alphanumeric word that normally describes a function. The index is either a digit or character that is used to identify an instance of data. For example, if a plugin can have these URLs captured: `index.html`, `#/employee/5/`, `#/employee/10/`, `#/employee/6/`, `#/employee/12/ChangePhoto` and `empDetails.html`. The syntax tree that will be generated to represent all states modeled by the tree on the left in Figure 32. To minimize the tree size and to avoid the need of representing every

single possible value of an index, the heuristic function combines and abstracts nodes that represent indexes accessing the same plugin. If the heuristic found repeated nodes (more than a specific threshold) that differ only in the index, then it infers that there is a pattern of calling the plugin. To represent the pattern, the nodes that represent indexes are replaced with a generic expression that represents the properties of the repeated indexes. Our implementation uses Java; so we used Java regular expressions to represent states. Indexes in number or character are replaced by “d+” or “w” so that when the states are traversed back, we get Java regular expressions to represent integer-indexed view state; for example, “#/employee/d+” represents parameterized view state #/employee in this scenario. The left tree in Figure 32 shows the resulted tree if the threshold is set to 3. Similar URLs are grouped as an expression. Extracted expressions can be used in the policy to validate subjects.

Algorithm 2 Build Call Syntax Tree Algorithm

```

1: procedure BUILDCALLTREEMAP()
2:   initialize Tree callTreeMap
3:   for each pluginKey in callMap do
4:     key = pluginKey
5:     initialize Tree pluginCallTree
6:     add root to pluginCallTree
7:     currentParent = root
8:     for all StateURL in callMap of pluginKey do
9:       if StateURL contains “#” then
10:        add child “#” child to parent currentParent
11:       end if
12:       tokenize StateURL by “/” into tokens
13:       add child tokens to currentParent
14:       update currentParent
15:     end for
16:     put into callTreeMap pair (key,pluginCallTree)
17:   end for
18: end procedure

```

For instance, if a plugin has these URLs captured: `index.html`, `#/employee/5/`, `#/employee/11/`, `#/employee/2/ChangePhoto` and `empDetails.html`. The syntax tree that will be generated to represent all states looks like the tree in Figure. 32. Similar URLs are grouped as an expression. The grouping decision is made based on the number of observed states that shares the same tokens except for a token that is either a number or a character, in that case, the token is replaced by the proper regular expression that represents the type of the token. Based on scanning the single-page based apps in our pool, we noticed that the hashbangs used follows this syntax:

$$(\#|\#/)A/ + [/A] * [/P] * [/A]*$$

Where A is for “action” which is an alphanumeric word and P is for “parameter” which is either a digit or a character. This syntax helped to found the approach that we have used to model the syntax of app states.

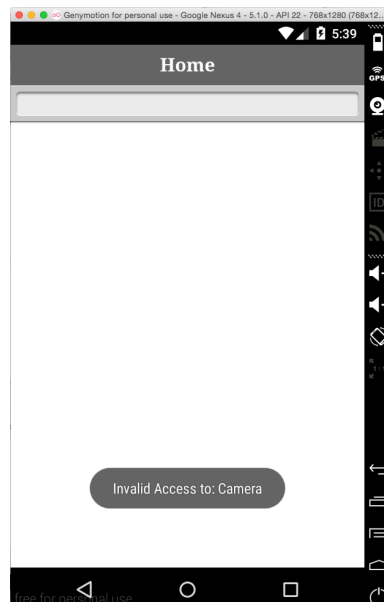
Enforce Stage This stage is meant to be when the app is released to the market. A fine-grained policy on the state level is generated based on the behavior that has been captured. Figure. (33(a)) shows a sample policy that can be mapped to a simple Access Control Rule syntax composed of `subject:(who can access)`, `object:(what resources to be accessed)` and `rights:(what operations are allowed)`. In this case, subjects are the “states” generated and objects are the plugins. The policy is represented by a parameter attached to each plugin declaration in the configuration file. Plugins without any `state` parameter cannot be accessed through any state of the app. Plugins added by default or by mistake, but are not actually needed in the app

```

<feature name="Storage" >
  <param name="android-package" value=".." />
</feature>
<feature name="Camera">
  <param name="android-package" value=".." />
  <param name="state" value="#/employee/\\d+" />
</feature>
<feature name="Contacts">
  <param name="android-package" value=".." />
  <param name="state" value="#/employee/\\d+" />
  <param name="state" value="empDetail.html" />
</feature>

```

(a) Policy Rules in config.xml



(b) Invalid Plugin Access

Figure 33: Plugin Access Policy

will have no state to be accessed from which will void the effect of including them. However, plugins that have a parameter attached to it will be executed in the states that conform to the value(s) indicated. For instance, in Figure (33) the Camera plugin is only allowed to be accessed in a URL that conforms to the regular expression “#/employee/d+” where the Contacts plugin can be also accessed from another page “empDetail.html”. Storage, on the other hand, cannot be accessed at all because it has not been captured by the reference monitor. Figure. (33(b)) shows how the app will react to an invalid call to a plugin according to the app policy. This is a home page where no plugin access is supposed to happen. We have injected a malicious script that tries to access the camera when on document load. However, Camera access is not associated with this state of the app; thus, it will not be executed and a Toast message is displayed to the user. The instrumented library generates this policy and updates the `config.xml` file of the app. The function `enforce()` that is added in the `exec()` function will use these rules to verify access after decrypting the state values using an assigned private key. If the access request complies with any rule, access is granted; otherwise, a `PluginException` is raised and user notification of an invalid access appears.

5.2.2 Behavior State Modeling

We propose an approach that enforces controlling transitions between different app screens based on the app behavior. The essence of the approach is very similar to the previous section in terms of monitoring app state transitions, creating a policy and then enforcing behavior to prevent any potential app logic manipulation. A behavior

is also defined by app interactions with other apps, such as Dialer, Messaging, and Maps. It is possible to call other apps the same way a link to a web page is called through using `<a href>` tag. For example `call this number`, enables the user to open the Dialer and pass a specific number when the link is clicked.

Similar to the process of app behavior elicitation approach discussed earlier, this is also a three-stage solution. The app should go through *Monitor* stage, *Generate* stage, then *Enforce* stage.

Monitor Stage: At this stage, any URL load change, DOM view transition, or external app call are monitored and saved. For single page DOM management, we extended the “router” function on JS side to extract current `window.location` value and log it. This log is handled in the native class `SystemWebChromeClient` where the URL is then saved to a database (DB) through the native side.

URL transitions to another page can happen either by navigating to another domain or loading the URL into the `CordovaWebView`. Capturing this transition can be done by monitoring `PluginManager.exec()` for the plugin `InAppBrowser`. We also monitor information such as URL and target host (*i.e.* `WebView` or `System Browser`).

The app calls to other apps through crafted URIs starting with `tel:` or `SMS`, for example, are monitored by adding a reference monitor in class `SystemWebViewClient`, specifically in the method `shouldOverrideUrlLoading` which gets triggered whenever a URL load happens.

Generate Stage This intermediate stage is necessary to generate an XML representation of the state machine notation for control abstraction. The DB contains logs of

URLs and page transitions in the sequence they were called. Then XML is generated to dictate the states the app have passed by.

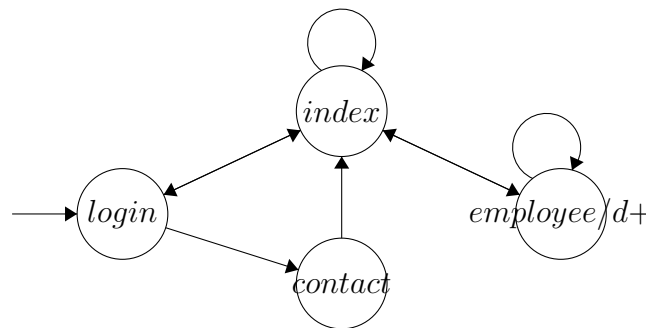
As for external apps, no policy is required. Instead, it is enough to generate navigations whitelisting only for those apps captured during the monitor stage. Figure. (34) shows a sample state-policy generated by our sample app. The representation follows

```

<urlstate id='login' value='login.html' type=start
location=local>
  <redirect id='index'>
  <redirect id='contact'>
</urlstate>
<urlstate id='index' value='index.html'
location=local>
  <redirect id='index'>
  <redirect id='login'>
  <redirect id='employeeview'>
</urlstate>
<urlstate id='employeeview' value='#employee/d+'
location=local>
  <redirect id='employeeview'>
  <redirect id='index'>
</urlstate>
<urlstate id='contact' value='https://www.abc.com/
contactus' location=external>
  <redirect id='index'>
</urlstate>

```

(a) App state machine XML representation



(b) App state diagram

Figure 34: App State Configurations

State Chart XML (SCXML) standard for representing the app control flow[69]. At this point, the implementation captures the state signified by `<urlstate>`. It also

captures 3 other features:

- If it is a start state, represented by the value of `type` attribute.
- If it is an external or local URL state, represented by the value of `location` attribute
- All the redirects (transitions) from the state to other states.

Developers need to copy this into their `config.xml` to be used for the *Enforce()* stage.

Enforce Stage A call graph is built based on the state machine configuration found in `config.xml`. Based on the example shown in Figure. (34(a)). A transition from one state to another is allowed only if the next state already exists as a destination from the current state, otherwise, the app cancels redirection. Our Java implementation of this graph is done using a `HashMap` of `states` and each state has an `ArrayList` to hold the adjacency list to other states for the corresponding state. Any URL change will require either `InAppBrowser` to be called or `onhashchange` to be triggered; thus, we added the check in two places. For URL loading using the plugin `InAppBrowser`, we added the check in the `PluginManager.exec()` which is as it has been already mentioned, a central hub for all plugin calls. For internal DOM routing, we have developed a Cordova API to handle redirects (`Redirectlist`). This plugin is very similar to Cordova's built-in `Whitelist` plugin. The API parses the `config.xml` to create a `HashMap` to use for checking. We use the trigger `window.onhashchange()` to check change in state for SPAs as shown in Figure. (35). To control the app interaction with external entities, the approach includes the navigation whitelisting settings. For example, the last two lines of configurations shown in Figure (34(a))

```

1 <script>
2   var oldHash = window.location.hash //global variable
3   $(window).on('hashchange', $.proxy(this.checkRoute, this));
4   checkRoute: function() {
5       var hash = window.location.hash;
6       var states = [oldHash, hash];
7       if(navigator.Redirectlist.checkTransition(states,
8         route(hash), route("error")));
9       oldHash = hash ;
10  }
11</script>

```

Figure 35: Check Redirection

includes only the Dialer and Browser to specific URL, instead of including *all* built-in apps and *all* URL (see Table 2).

5.2.3 Performance Analysis

Performance overhead has been associated with dynamic code analysis as a disadvantage. This section describes performance measurements of different aspects of the proposed instrumented library compared to the stock version of Cordova library (*version 6.x*). We have used a device with the following specifications: Model Moto G (2nd Generation), Android *version 5.0.2*, Internal Storage 5.5 GB.

I. Enforce Time vs App Size The purpose of this test is to measure the scalability factor of enforcing plugin access policy on a varying number of pages/states of a hybrid app. For this test, we use one plugin (Compass) with 10 apps having 1 to 10 pages/states. We choose the *Compass* plugin because it is a plugin that requires a sensor access and at the same time has a **frequency** option that we can use to enable automatic frequent calls from the code. We have also chosen to test up to 10 pages because the average ratio of page to app based on the apps pool [61] was 10 to 1.

Hence, we conducted a series of tests measuring the time of enforcing a plugin access on 10 apps having 1 up to 10 pages/states. The time needed by checking the policy is the time between a plugin request and the decision to grant or prevent plugin access. This interval is measured for every single plugin access in every page/state and then the readings are averaged, Figure. (36) shows that checking the policy varies between 0.4 to 0.58 milliseconds. The trend line shows that the time is linearly increasing with a positive slope value of 0.021. Although the app size and policy check time are positively proportional, the number of pages/states does not seem to be a strong factor of increasing the time given the low slope value.

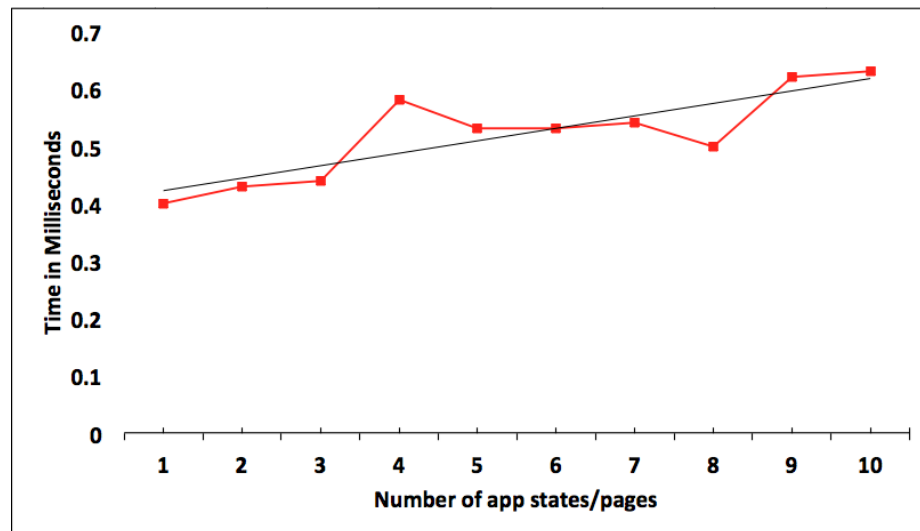


Figure 36: *Enforce Time vs # States*

II. Original vs Modified `exec()` Time *with* Plugin Type The purpose of this test is to check the relative overhead added by enforcing a policy related to a specific plugin and executing it compared to the time needed by the stock version. Another aspect of this test is inspect how different plugin types perform in both cases.

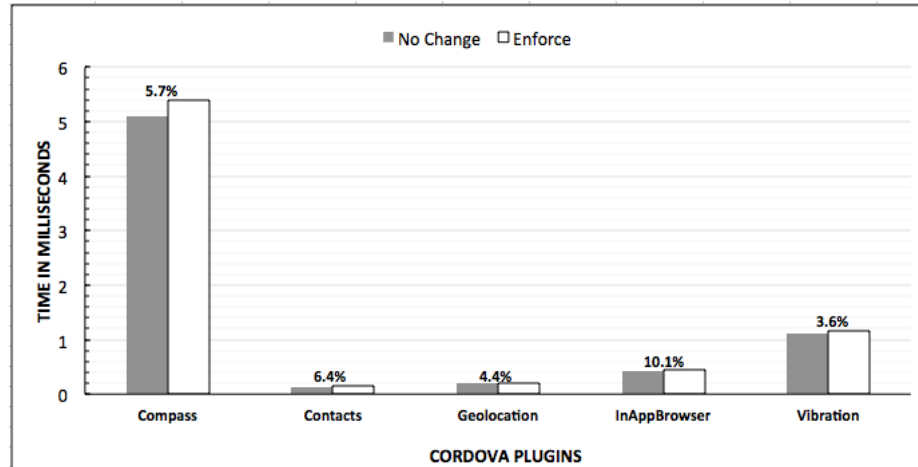


Figure 37: `exec()` with Plugin Type

We include only “*Enforce*” because the “*Monitor*” stage is an instrumentation of the regular app, where data are being collected and saved and it is done temporarily while the app is being developed/tested. While the “*Generate*” stage is an intermediary stage where all the rules are generated once. “*Enforce*” is the stage of the app when released to the market, hence, our performance testing focuses on comparing the overhead of an app in the *Enforce* stage with an app using the stock library (*version 6.x*). As Figure. (37) indicates, a set of plugins selected that represent a variety of plugin categories. *Compass*, *Geolocation*, and *Vibration* are plugins that access the device sensors. While *Contacts* is a plugin that accesses the device contacts database and finally the *InAppBrowser* which does not access any device specific feature but is designed to host web content in a container other than the *CordovaWebView*. This plugin also is used in our tool for controlling external URL loads. The experiment shows that the time needed by the Modified `exec()` (with *Enforce*) relative to the stock version varies between 3.6% for *Vibration* to 10.1% for *InAppBrowser*. The type of the plugin does not seem to affect the relative overhead time.

III. Redirect Plugin Check Time As for internal DOM manipulation and URL redirects, we have measured the time it takes a router function to decide the view and render it in 2 cases, regular routing and routing after checking the `RedirectList` plugin. Averaging the time of 100+ runs of both cases revealed the following: Routing between views without checking takes only ≈ 10.8 microseconds while after checking using `RedirectList` plugin takes ≈ 50.6 microseconds which can be attributed to the roundabout time of messages between JS and Native side.

CHAPTER 6: CORDOVACONFIG: AUTOMATED TOOL FOR CONFIGURING HYBRID APPS

We explained securing Hybrid Apps through a Behavior-Based configuration model. Automated behavior modeling is essential to seamlessly assisting developers to control app behavior through aligned configurations. The number of developers involved in the development of mobile apps reached 12 million in 2016, according to Evans Data. This number is more than half of the total worldwide population of 21 million developers. It is expected to exceed 14 million by 2020 [26]. With this massive shift in demand of labor force toward mobile apps development, the need to support developers with tools to help build secure apps is essential.

The current distribution of the library suffers several security limitations related to its configuration scheme. Mainly, having coarse-grained settings and risky default values. Code injection attacks are the most prominent threat to hybrid apps [41]. Malicious injected code can abuse the bridge provided by the library to access device native resources (plugins) such as camera, geolocation, and contacts. Malicious code may also target the app itself as it can maliciously tamper the app logic by redirecting to an unwanted page. Table 7 summarizes how improper configurations can maximize damage resulting from code injections attacks. In their latest report of mobile top 10 risks, OWASP identified “Improper Platform Usage” as the number one risk in mobile apps development. This includes misuse of a platform feature or failure to use

Table 7: Configurations Issues and impact on apps' security

| Configurations Issues | Impact |
|------------------------------|--|
| Not having a proper CSP | Enable malicious code to be triggered/ executed |
| Global plugins declaration | - Plugins can be accessed in all app pages/states - Having unnecessary plugins (over-privileged) |
| Risky defaults | - Resource network access is set to "*" <ul style="list-style-type: none"> - Allow calling built-in apps such as SMS, TEL, Maps |

platform security controls resulting in an easy exploitability and a severe impact [21].

Smartphone apps are not as trusted as web nor desktop applications. Research [23] shows that users are more concerned about privacy on their smartphones than their laptops and they are more apprehensive about performing privacy-sensitive and financial tasks on their smartphones than their laptops. Improving smartphone apps security and privacy cannot be achieved without involving the app developer in the process.

The lack of application security skills, tools and methods are ranked as being one of the top three challenges faced by developers to maintain apps' security [39]. The lack of skills, tools, and methods induces the need to provide training and increase awareness among developers to help implement better development practices.

Coding derives a cognitive burden on developers, mobile apps' development is no exception. Productivity bottlenecks divert efforts to repairing errors rather than learning to avoid them. Developers are constantly under severe time pressure; therefore, they seek the fastest ways to get the job done.

Integrated Development Environments (IDEs) and tools support have been playing a vital role in helping developers avoid errors and improve productivity. Nonetheless, certain types of logical errors resulting from poor coding practices such as misconfig-

uration and not following platform recommendations, are as harmful as other syntax and semantic errors. This category of errors may cause security breaches. Previous research [75] mentioned that there is a disconnect between developers' conceptual understanding of security and their attitudes regarding their personal responsibility and practices for software security. Many developers regard configurations as non-functional and not as important as the core function of the code, not to mention that the impact of such flaws may not necessarily interfere with the program logic but only surface in the course of security breaches.

In 2008, Phonegap or Cordova, the leading mobile hybrid platform, was introduced to the community as a solution to several challenges facing the mobile development industry such as market fragmentation and the high cost of development and training. Hybrid mobile platforms can target several operating systems using the same code base. The platform is supported with libraries that implement the bridge that enables access device specific features using Javascript. Cordova Library is a middleware that is a common component in many popular hybrid platforms such as PhoneGap, IBM Worklight, App Builder, Sencha, Monaca, and Appery.io. This component is the real enabler of connecting the two worlds (Web and Native) inside a hybrid app. Recent statistics in Google Play show that hybrid apps constitute a 5.84% of the market share. Some apps are able to attract a customer base of 10,000,000+ [15]. Business, Medical, and Finance apps are at the top of the list of hybrid apps.

Adoption of the technology has faced some challenges due to the lack of tooling support [55]. Later, several platforms have invested not only in providing tooling support but also in incorporating other technologies, such as cloud-based builds. Most of the

support is toward hard-core coding, debugging, and deployment. App configuration support is not only a low priority but also suffers limitations such as risky default settings and a basic XML based interface. The current platform adopts a command line interface to interact with the app. The app configuration can be changed by editing a text file (`config.xml`) that contains xml formatted settings.

In this work, we are proposing CORDOVACONFIG. A web-based tool to help developers configure their apps securely based on the app behavior. The tool aims to minimize the burden on the developer by providing generated configurations based on what has been monitored. The tool does not only generate configuration settings, but also plays an educational role by explaining each configuration setting meaning and consequences. Educating developers on secure programming techniques through the IDE proved to be effective means of raising awareness among developers [73][58][74][79]. Generating configurations values based on observed app behavior is essential to remove some burden from developers shoulders and help control app behavior at the same time.

We have developed CORDOVACONFIG, an interactive web-based tool that allows interaction with developers to configure hybrid apps. The tool is built on the top of the instrumented cordova library mentioned in section 5. CORDOVACONFIG walks the developer in a wizard style where there are different stages of app configuration to finally generate what the developer determines to be a valid behavior. Since the tool's feed of information about the monitored app behavior is based on dynamic analysis, false negatives are presumed. Involving the developer is necessary not only to eliminate false negatives but also to give the developer more control to tweak the

configurations in a simple way. The tool also presents an educational part at the beginning where developers get explanations of the meaning of the configurations and the impact of keeping the current values. The developer can download the configuration file and use it instead of the default one. Figure 38 explains the tool work flow. The tool directs the developer through several steps before generating configurations.

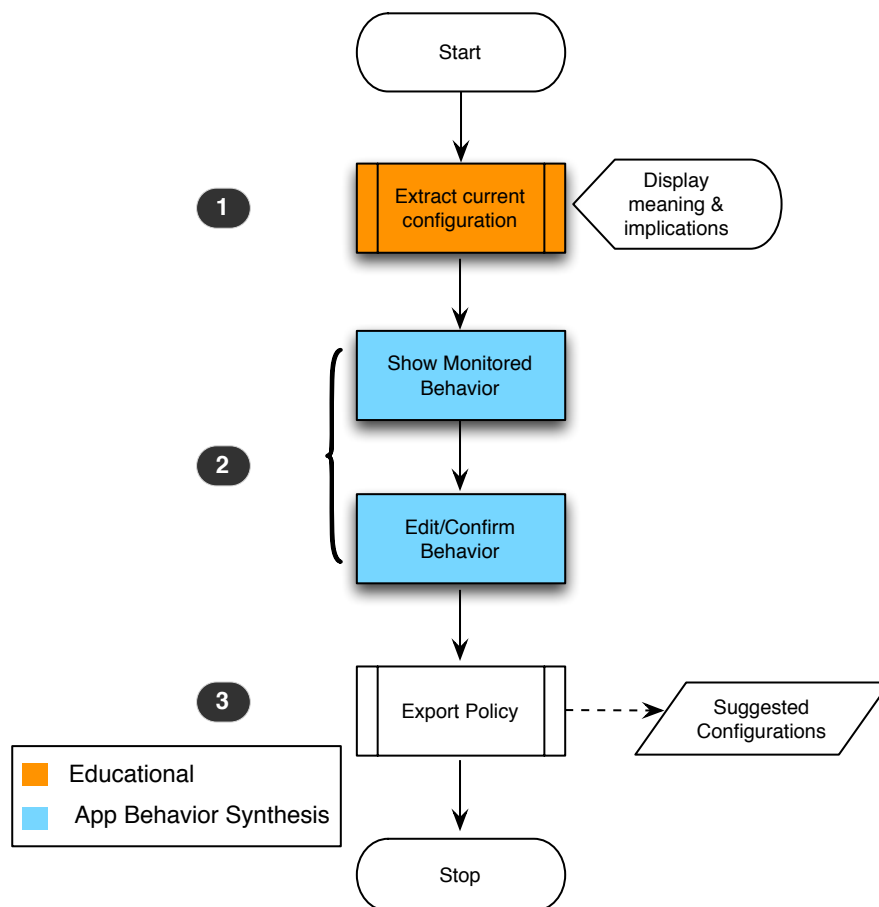


Figure 38: CORDOVACONFIG Work Flow

6.1 Educational Goals

Educating users is a pillar of usable security. Thus, we have designed the tool to help improve developers' understanding of configuration items. The tool starts with reading the current configurations in the `config.xml` file. Then, it generates an explanation in terms of the meaning of each configuration item and the impact of having these configuration values in the app. For instance, Figure 39 demonstrates

Your App Configuration Analysis - Part 3

Network Requests (images, JS..etc) are allowed to be made:

```
1 <access origin="*" />
```

Which means accessing:

- Don't block any request

URLs the WebView can be navigated to:

No configuration found!

★ By default, WebView is allowed to navigate to local files only

Android Permissions accesses are:

```
1 <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
2 <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
3 <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
4 <uses-permission android:name="android.permission.READ_CONTACTS" />
5 <uses-permission android:name="android.permission.WRITE_CONTACTS" />
6 <uses-permission android:name="android.permission.GET_ACCOUNTS" />
7 <uses-permission android:name="android.permission.RECORD_AUDIO" />
8 <uses-permission android:name="android.permission.RECORD_VIDEO" />
```

Which means accessing Android System:

- Location – An app can use the device's location. This includes two different location settings, approximate and precise. Using the approximate location option, the app gets the device's location from the network, and the precise option uses the device's GPS and network to determine the location. To do this, the permission allows the app to access extra location provider commands and GPS.
- Photos/Media/Files – The application has the ability to use the file on the device with the application installed. This includes reading and writing to the SD card and USB storage. The app can also mount and unmount external storage as well as format external storage. This permission deals with reading external storage on newer devices
- Contacts- find accounts on the device, see and modify the owner's contact card and add or remove contacts from the device
- Video/Audio- capturing and encoding a variety of common audio/ video formats integrated into your applications

[Next](#)

Figure 39: Explaining scanned configurations meaning/impact

a set of configurations (including resource network access and navigation whitelist) configuration found in the `config` file and the corresponding meaning/impact of such value. It also captures the android permissions found in the `AndroidManifest` file and explains the consequences of using such permissions. There exist several parts (pages) of this educational phase to go over all the configurations values.

The goal of this part is to raise awareness of cordova based apps' configurations meaning and their impact. Figures 54,55 and 56 in Appendix A show sample screenshots

of current configuration explanations.

6.2 App Behavior Synthesis Phase

6.2.0.1 App States

As explained in the previous section, the behavior-based model captures app states. Each app state is associated with a set of plugin access(es). Each state is also captured

App State 2 - (S1):

#employees/ad+ State Identifier

Access the following plugin(s):

- Geolocation.
- Camera.
- Contacts.
- Connection.

Plugins Access

[Add More Plugins?](#)

Captured screenshots of current state

| Back | Details | Back | Details | Back | Details | Back | Details | Back | Detail |
|-----------------|--|-----------------|--------------------------------------|-----------------|--|-----------------|--|-----------------|---------------------------|
| | Michael Scott Regional Manager | | Pamela Beesly Receptionist | | Meredith Palmer Supplier Relations | | Phyllis Lapin Sales Representative | | Ryan H Vice Pre |
| Call Office | 570-123-4567 | Call Office | 570-999-7474 | Call Office | 570-083-8167 | Call Office | 570-632-1919 | Call Office | 212-999-8887 |
| Call Cell | 570-865-2536 | Call Cell | 570-999-5555 | Call Cell | 570-588-6567 | Call Cell | 570-241-8585 | Call Cell | 212-999-8888 |
| SMS | 570-865-2536 | SMS | 570-999-5555 | SMS | 570-588-6567 | SMS | 570-241-8585 | SMS | 212-999-8888 |
| Add Location | Add Location | Add Location | Add Location | Add Location | Add Location | Add Location | Add Location | Add Location | Add Location |
| Add to Contacts | Add to Contacts | Add to Contacts | Add to Contacts | Add to Contacts | Add to Contacts | Add to Contacts | Add to Contacts | Add to Contacts | Add to Contacts |
| Change Picture | Change Picture | Change Picture | Change Picture | Change Picture | Change Picture | Change Picture | Change Picture | Change Picture | Change Picture |

Page(s) apply to this pattern require(s) the following permission(s):

- ACCESS_COARSE_LOCATION.
- ACCESS_FINE_LOCATION.
- ACCESS_LOCATION_EXTRA_COMMANDS.
- WRITE_EXTERNAL_STORAGE.
- READ_CONTACTS.
- WRITE_CONTACTS.
- GET_ACCOUNTS.
- INTERNET.
- ACCESS_NETWORK_STATE.

OS permissions needed

Agree?

[Add More App State/ Plugin Accesses?](#)

Figure 40: Plugin accesses and OS permissions captured *per* state

with a screenshot of the app when using the plugin. A list of plugins accessed, URLs accessed, and the corresponding Android permissions needed to be granted for these plugins are all captured per state, see Figure 40 . The developer has the option to: add any plugin that has not been captured or delete any unnecessary access to a plugin. The tool also enables the developer to define a new state. Figures 57 and 58

in Appendix A show sample app states captured.

6.2.0.2 App States Transitions

The tool captures the app state transitions and translates them into configurations to control app flow against attacks trying to tamper the app control flow. The captured states transition is displayed to the developer in the form of a state diagram (see Figure 41). Approving it creates configurations in a State Chart XML (SCXML) format that represents the app flow. A developer can change the configurations

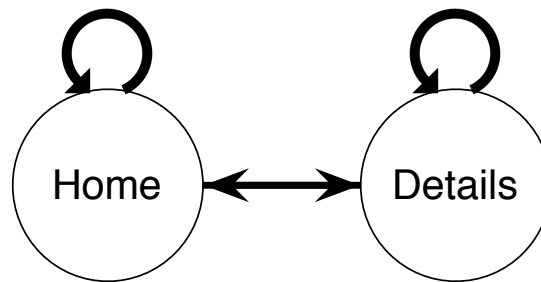


Figure 41: State Transition Example

generated to any other form she chooses to. Identifying app states and their transition may help the developer better understand the app behavior and the new configuration scheme suggested. Figure 59 shows a sample app transition diagram.

6.2.0.3 App interactions with external entities

Another important aspect of modeling the app behavior is to capture the app interactions with external entities, such as:

- Other Applications (**Intents** in Android).
- Remote network resource access
- Navigation URLs within the app

App Behavior Confirm - Part2:

In this part, you will see the how your app interacted with external entities/ resources

Your app have accessed the following resources from external websites :

[http://\[REDACTED\]/img](http://[REDACTED]/img) Captured URLs of resource access

Add More External Access Origins?

Your app invoked these built-in apps? (check is you want to add more)

System Messaging App Captured External Apps Calls

System Dialer

System Email

System Maps App

Your app WebView navigated to these domains: Captured URL navigations

(*By default, Webview and iframes navigations only to file:// URLs. There is no non-http schemes allowed.)

Allow to navigate to external URLs?

Confirm Behavior

Figure 42: App interaction with external entities

Figure 42 shows a sample of the settings related to interactions with external entities.

A developer can also add/modify/delete any of these settings then confirm them.

6.3 Generating Configurations & Permissions

After the developer confirms the behavior, configurations are generated to align with it. As shown in Figure 43, generated settings are comprised of three parts:

- Configurations to control plugins, URLs, Navigations and interactions with other apps per state.
- Configuration for app state transition, which is a new configuration item added by our tool to control app transitions.

- Android permissions per state, which can be added to the `Android-Manifest.xml` file to control the native side of the app.

Our prototype implementation supports interactive environment for developers to help further customize the configurations in case the observed behavior did not capture all required features.

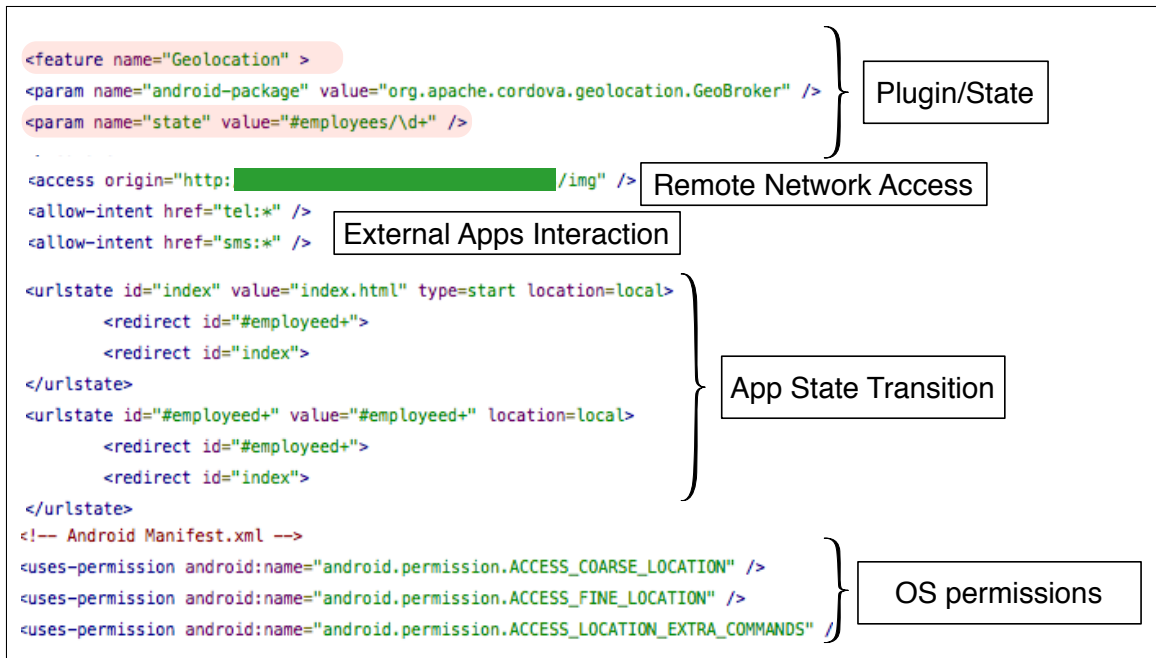


Figure 43: Generated Configurations

CHAPTER 7: USER STUDY: HYBRID APPS DEVELOPERS PERCEPTIONS

The current CORDOVACONFIG prototype is a proof-of-concept web-based tool for generating configurations based on the observed behavior of an app. The goal of this tool is to allow interaction between the developer and the tool to address an important and often overlooked step, that is app configurations.

The main goal of the user study is to explore developers perception of the tool and its value. A controlled repeated measure study is conducted to either prove or refute the following hypotheses:

- H1: Using CORDOVACONFIG changes developers' mental model in regard to understanding the boundaries of the required privileges of an app.
- H2: Using CORDOVACONFIG improves developers' understanding of a specific app' configurations.

The study also further explore the following questions:

- Q1: What is developer's' perception of risky settings
- Q2: What is developers' perception of the value of CORDOVACONFIG

In order to test the hypotheses, we identify the variables and the research instruments used to measure them (see Table 8). The user study is a controlled repeated measure experiment. Metrics for H1 and H2 are set to be the score of the correct answers in

Table 8: Variables Measurement Methodology

| Variable | Research Instrument | |
|--|-------------------------------------|-----------------------|
| Understanding of an app configurations | Pre/Post Survey | Questions - 1,2,3,4,8 |
| Changing developer's mental mode | Pre/Post Survey | Questions - 5,6,7 |
| Tool usability | System Usability Scale (SUS) Survey | |
| Perception of risky settings | Interview | Question 3 |
| Perception of the tool value | Interview | Question 2 |

the Pre/Post Survey. Tool usability is also measured using SUS index [67]. The last two questions aim to explore unknown aspects, so we use qualitative data points to understand the current situation of developers' perceptions.

7.1 Case Study App

The app presented in this study is a simple Employee Directory application. It is referred to as `empDir`, which views information about a list of employees through a summarized list item. Clicking on a list item displays a detailed information about that employee. It also enables the user to perform the following functionalities:

- Getting the current location of the device. This functionality requires access to the Geolocation API.
- Changing the employee profile photo. This functionality requires access to the Camera API.
- Saving the employee contact information into the device contacts list. This functionality requires access to the Contacts API.

The app (see Figure 44) is taken from one of PhoneGap tutorials[25]. This app is chosen for the following reasons:

- Utilizes several APIs such as Geolocation, Contacts, and Camera.

- Interacts with other Apps (Dialler & Messages)
- Loads resources from an external URL.

This single page app makes a good candidate for an app that needs to customize configurations for almost all configurations items found on `config.xml`. It also utilizes the state identification and aligned configuration per state. The app starts with home

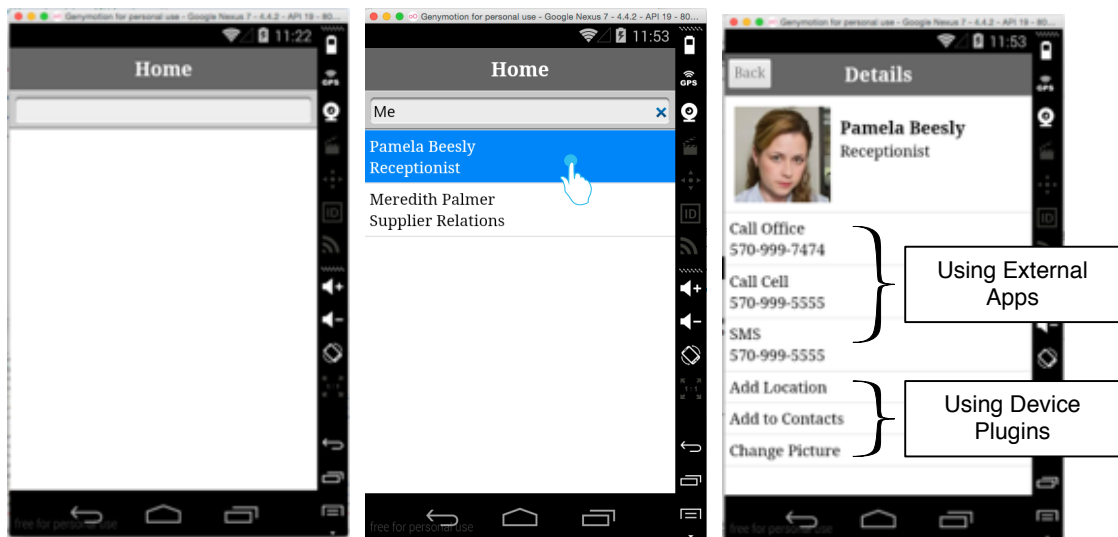


Figure 44: Employee Directory App

page that has a search text. The user can enter the name of the employee which will filter the list of the employees to match the text. Once the user selects an employee, a detailed information page opens. The page enables contacting the user through calls and SMS. It also enables adding a location for the user, adding the employee phone numbers to the device contacts and changing the current picture of the employee using the device camera.

7.2 Recruitment

We have recruited 22 students who have a background in mobile application development and Cordova/PhoneGap development. We have used flyers and emails to call for participation. Participants are rewarded \$20 gift cards in return of their time and effort. Most of the participants are either current or previous students of a Mobile Application Development course focusing on Android. In addition, the course also offers material focusing on hybrid apps development by introducing students to PhoneGap/Cordova library. The assignments given to the student related to this subject focuses on familiarizing students with using cordova built-in plugins. In addition to other web-based platforms specific to mobile apps.

7.2.1 User Study Protocol

The participants are provided with the app `empDir` that is mostly implemented missing only the third functionality -adding to contacts. Participants are asked to resolve what is missing through testing the app and checking the app files. The app configurations are kept to default settings. Once the participant is able to detect the missing functionality. She is asked to implement it. On average, each participant took 45 minutes to finish this step. To be able to finish the task, participants need not only to write the code for the missing functionality -calling plugin API on JS side- but also to configure the app properly so that it includes the declaration of the contacts plugin API. The goal is to familiarize the participant with the app code and its logic. The participants are asked to test the app and check if it satisfies all requirements. At this point, participants experienced a typical hybrid app

development cycle. Then participants are asked to answer a pre-survey. After that, participants are provided with the same app built on the instrumented version of the library. Thus, all app transactions are monitored. Participants are asked to test this app to make sure it's working properly, meanwhile, the tool can have more data on the app behavior by logging all app interactions and transitions. Then, participants are asked to use `CORDOVACONFIG`. After that, participants are asked to answer a post survey. Finally, participants are interviewed. The steps of the procedure can be summarized as follows:

1. Complete the work on `empDir`.
2. Answer Pre-Survey
3. Test `empDir` on instrumented cordova library
4. Use `CORDOVACONFIG`
5. Answer Post-Survey
6. Interview

To cancel any bias, two versions of Pre and Post surveys (Pre/Post Survey A & Pre/Post Survey B) are formulated such that questions in Pre-Survey A are used on Post-Survey B and vice versa. Participants are randomly assigned to answer either Pre/Post Survey A or Pre/Post Survey B. Both surveys aim to measure the same parameters but through different questions.

The interview at the end of the study is needed to measure qualitative variables such

as developer perception of the tool and their awareness in hybrid apps' security. Each interview lasted 5 to 15 minutes. We began by asking participants if they would be interested in using this configuration tool and what is the value of such a tool. Next, we asked our participants to explain what would be the implications of leaving the default configuration on the users of the app and how the generated configurations are different. Then, we asked the developers if the tool was confusing at any point during their interaction. We also asked participants whether they thought they would likely to pay attention to default configurations and fix risky setting without such tool support. Finally, we asked the participant whether s/he are familiar with Content Security Policy and what it stands for.

7.3 Using CORDOVACONFIG

We have created a shortcut for the tool URL on the computer desktop. Each participant is asked to click on the icon to start the tool. The average time participants spent using the tool is \approx 15 minutes. Only 2 participants asked questions while using the tool. The questions were mainly about how to use the tool.

The final output of the tool is the generated configurations. All participants indicated that they would use the generated configurations and the Android permissions rather than keeping the old ones, mainly because they indicated that the configurations are precise and exact and conform with the app functionality.

To observe how much attention participants would pay to configurations when using the conventional method, we have changed the configurations of the app. We have added unnecessary plugins but also removed the needed plugins for the app to

function. By the end of the development testing, we have noted that all participants added the required plugins, but NO ONE has removed the unnecessary plugins. This was a strong signal that developers tend to change configurations by adding the required settings, but rarely they do any checking or revision of the setting values as long as the app is working.

7.4 Results

Below a detailed demonstration of the participants' background, data analysis procedures, and interpretation of research results.

7.4.1 Participants Demographics

Education level of participants varied between graduate 18 (81.8%) and undergraduate 4 (18.2%) aging between 21 - 32 years old. Most of the participants are male comprising 90.9% (20) as opposed to 9.1% (2) female participants. Table 9 shows a

Table 9: Participants Development Experience

| Experience\# Years | <1 | 1-3 | 3-6 |
|----------------------------|--------------|------------|------------|
| Mobile Based | 68.2% (15) | 22.7% (5) | 9.1% (2) |
| Web Based | 31.6% (7) | 54.5% (12) | 13.6% (3) |
| Mobile Hybrid Based | 95.5% (21) | 4.8% (1) | - |

summary of the participants development experience in three areas; Mobile Applications, Web Applications, and Mobile Hybrid Applications.

As this tool is designed to maintain app's security through aligned configurations, we have asked the participants to self-assess their level of security knowledge (Novice, Intermediate, Expert). Most of the participants 86.4% (19) have indicated a Novice

level while 13.6%(3) have indicated an Intermediate level.

Overlooking security and dealing with it a secondary issue is a common practice

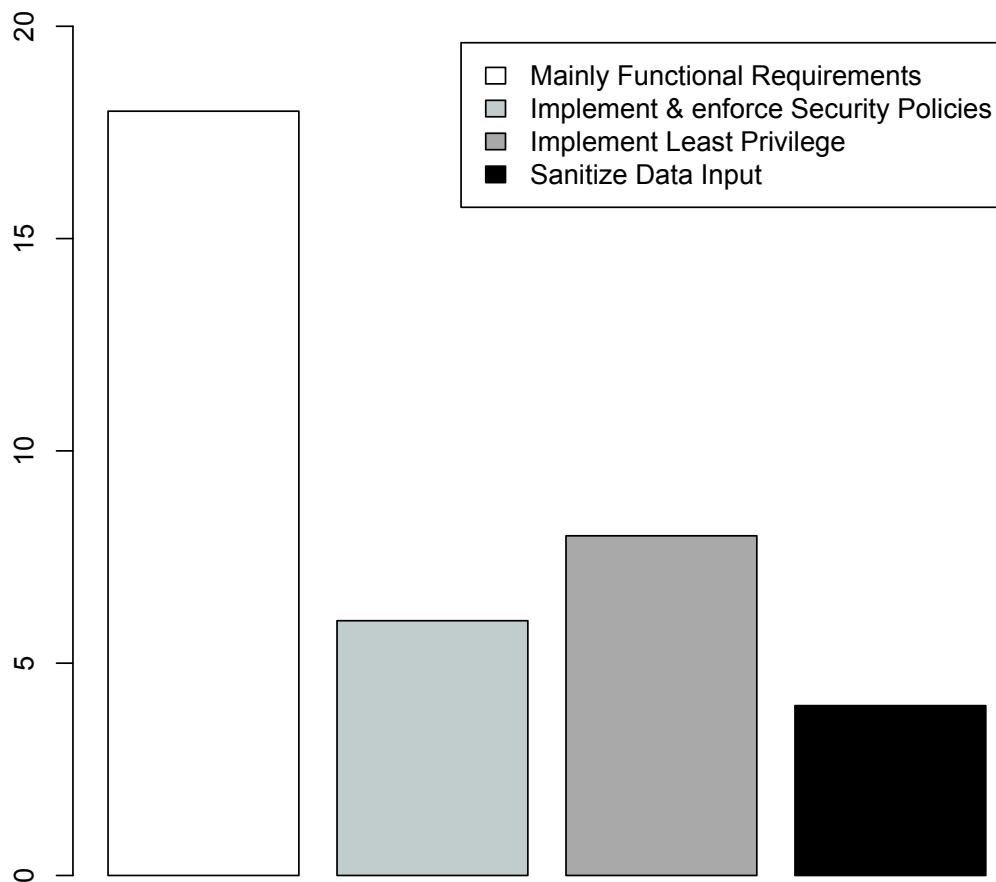


Figure 45: Common Coding Practices followed by participants

among developers. To check if this also applies to this group, we asked them to check what applies as a common coding practice. Figure 45 depicts the options and shows that, as expected, they normally focus on the functional requirements of the app more than any other security-related practice.

7.4.2 H1:CORDOVACONFIG & Configurations Understanding

Participants answered a set of questions before and after using the tool. The questions focus on measuring developers understanding of configuring the app to satisfy specific requirements. Specifically, settings related to :

- Plugin usage
- Network access
- Platform permissions
- Interaction with external apps

Participants are also asked to provide an explanation of certain settings impact, and how-to change configurations to satisfy a specific need. The answers to these questions are collected. The average of scores before using the tool is 3.4 while it changes to 4.4 after using the tool. Figure 46 demonstrates participants' score in the Pre/Post survey questions related to this variable. We can see that most participants are able to score higher in the post survey. Nonetheless, there are 4 participants who had a negative effect.

A dependent t-test (within-subjects) was conducted using participants' scores before ($M=3.4$, $SD=1.47$) and after ($M=4.4$, $SD=0.79$) the survey. The result supports the hypothesis that developers are more likely to understand the purpose and the meaning of the app configuration items after using this tool, $T(21)=7.96$, $p=0.01$, $p < .05$.

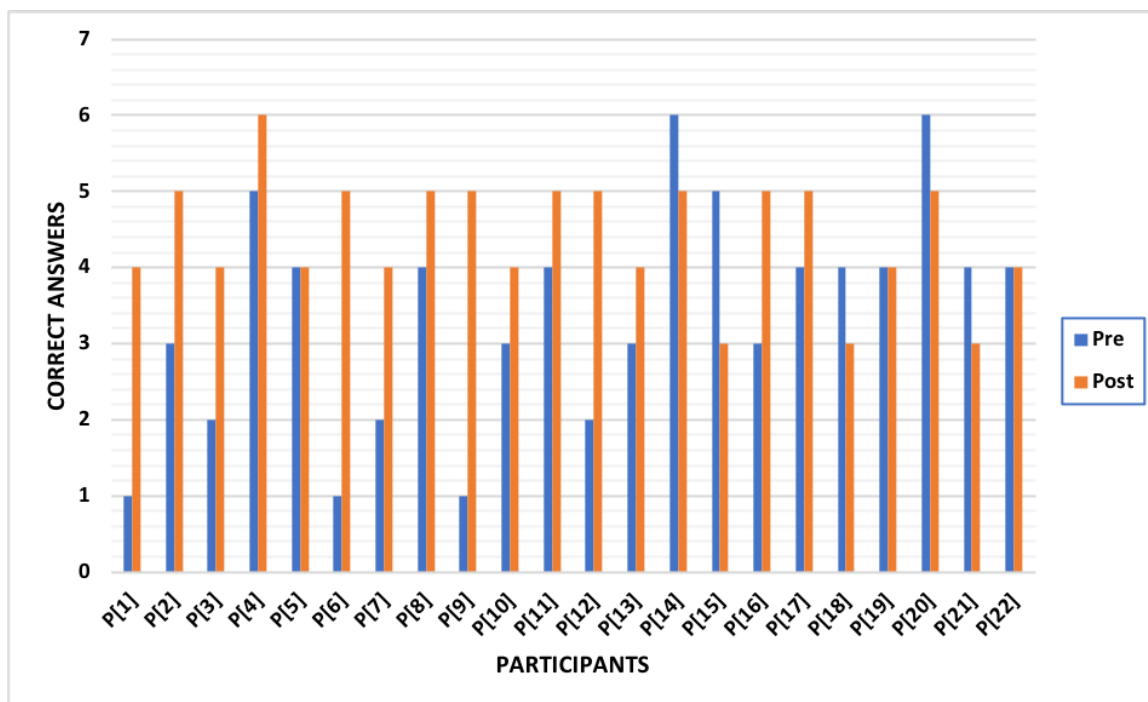


Figure 46: Developers' Configuration Understanding Scores

7.4.3 H2:CORDOVACONFIG and Developers Mental Model

One of the features provided by most hybrid platforms together with Cordova, is to allow inter-applications interactions between apps. To perform an operation that requires calling another app, the developer need to use URIs. For instance, to dial a number developer can use this URI:

```
<a href="tel:1(xxx)xxx-xxxx">Call Me!</a>.
```

Clicking this link launches the native Dialer app and passes the number. Other native apps such as maps, SMS, and mail can be called the same way. Default configurations allow interaction with all these apps.

Favoring simplicity at the expense of security might be the reason why default configurations are kept this way. However, it is important to recognize that this violates the

principle of least privilege. Moreover, it may invite a misconception at the developer end in terms of understanding the boundaries of the hybrid app. Developers need to recognize that having maps functionality in the app - for example - may require calling the built-in maps app but this should not require the app to have any permissions related to maps functionality. Developers need to recognize that once the maps app is open the control is transferred to the maps app which is outside the boundary of the app. Developer needs to understand that they do not need to include any platform permission related to geolocation as this is outside the boundary of the app. Thus, in the survey, we ask questions to test the tool effect on changing the developer mental model by asking what permissions, configurations and changes needed to call native apps. The questions addressed the case when the app calls another app and when the app itself uses a functionality and how this should differ in terms of what permissions to ask for. The average of scores before using the tool is 0.6 while it changes to 1.22 after using the tool. Participants scores are shown in Figure 47.

A dependent t test (within-subjects) using participants' scores before ($M= 0.5 ,SD=0.7$) and after ($M= 0.1 , SD=0.8$) the survey was performed. The result supports the hypothesis that developers are more likely to understand the boundary of the app and to recognize the needed permission/configurations needed, $T(22)=4.44, p= 0.047, p <.05$.When we categorized participants based on common coding practices they follow (Figure 45), we have a found two groups have significant relationship with this variable. Namely participants who checked that they usually implement and enforce security policy ($T(1,5)=25, p= 0.004, p <.05$) and participants who checked that they would sanitize external input ($T(1,3)=27, p= 0.014, p <.05$).

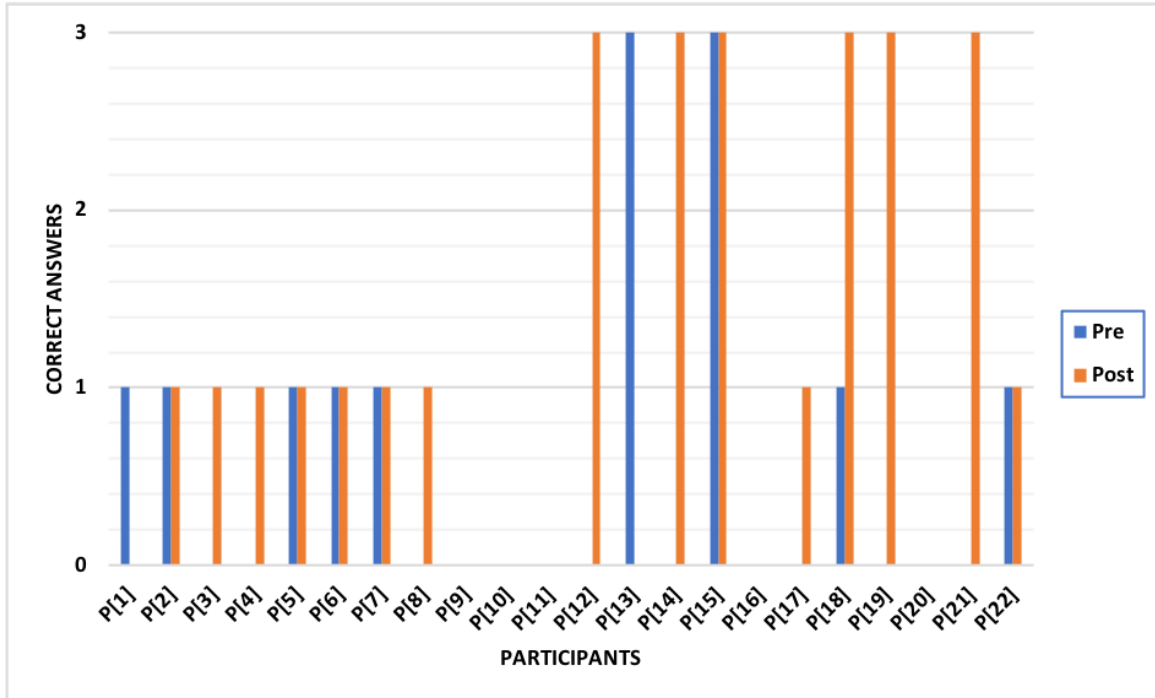


Figure 47: Developers' Mental Model Change Scores

7.4.4 Q1: Developers' awareness of potential risks

The goal of using the tool is to generate a set of aligned and fine-granular configurations rather than using the default configurations. To examine the difference the tool is making developers are asked to compare the configurations generated by the tool and the default configurations of `empDir`. Differences include :

- The inclusion of only the plugins that are necessary.
- Setting specific values of allowed URLs for resources' loading rather than the default value '*' that allows all domains
- Setting specific values for types of apps allowed to be launched from the app

When asked, most participants have noticed the first two. Then when they were asked to indicate the implications of keeping the default configurations in the app rather

than using the generated configurations, they gave different answers which show the different perspectives of understanding. Figure 48 and 49 show the distribution of developers answers. Although most developers are able to indicate a security

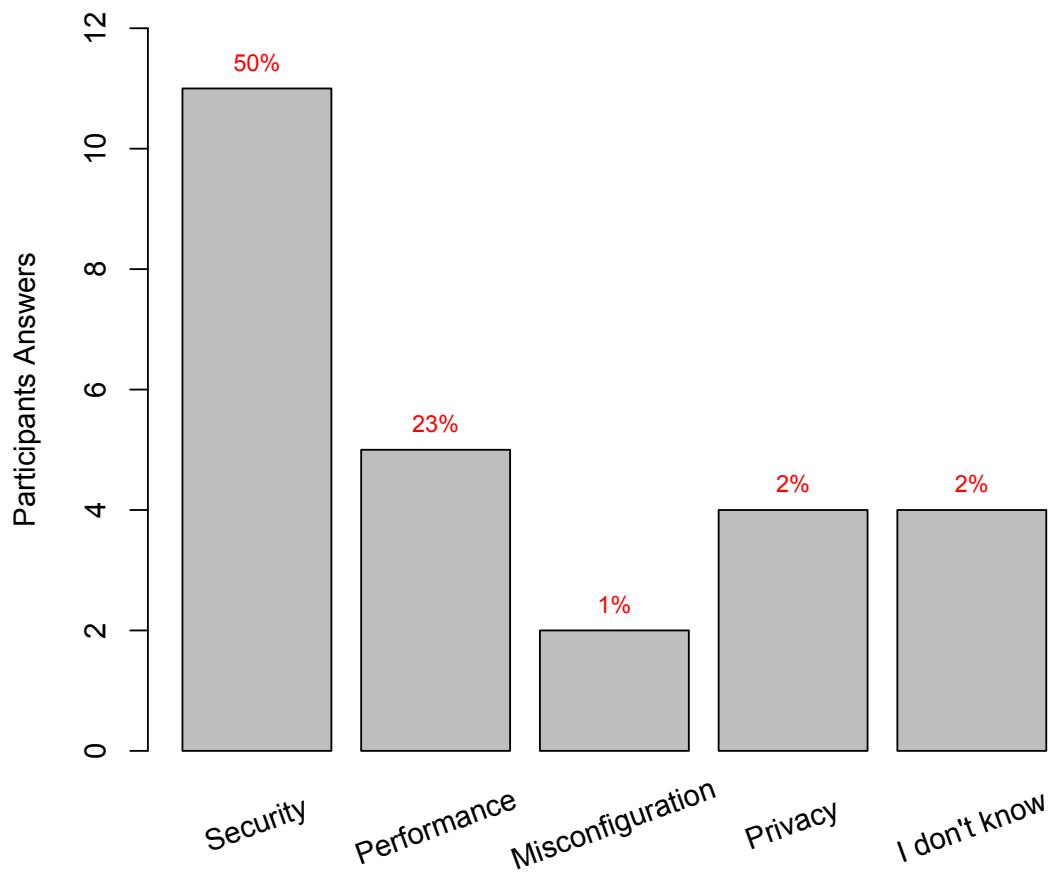


Figure 48: Perceived Implications of having unaligned plugins settings

related risk of keeping the default configuration, only *four* participants were able to mention attack types that can potentially compromise an app. The rest were unable to describe nor mention a scenario that can put an app under risk.

Previous work [18] postulates three reasons for “Why Good People Write Bad Code”:

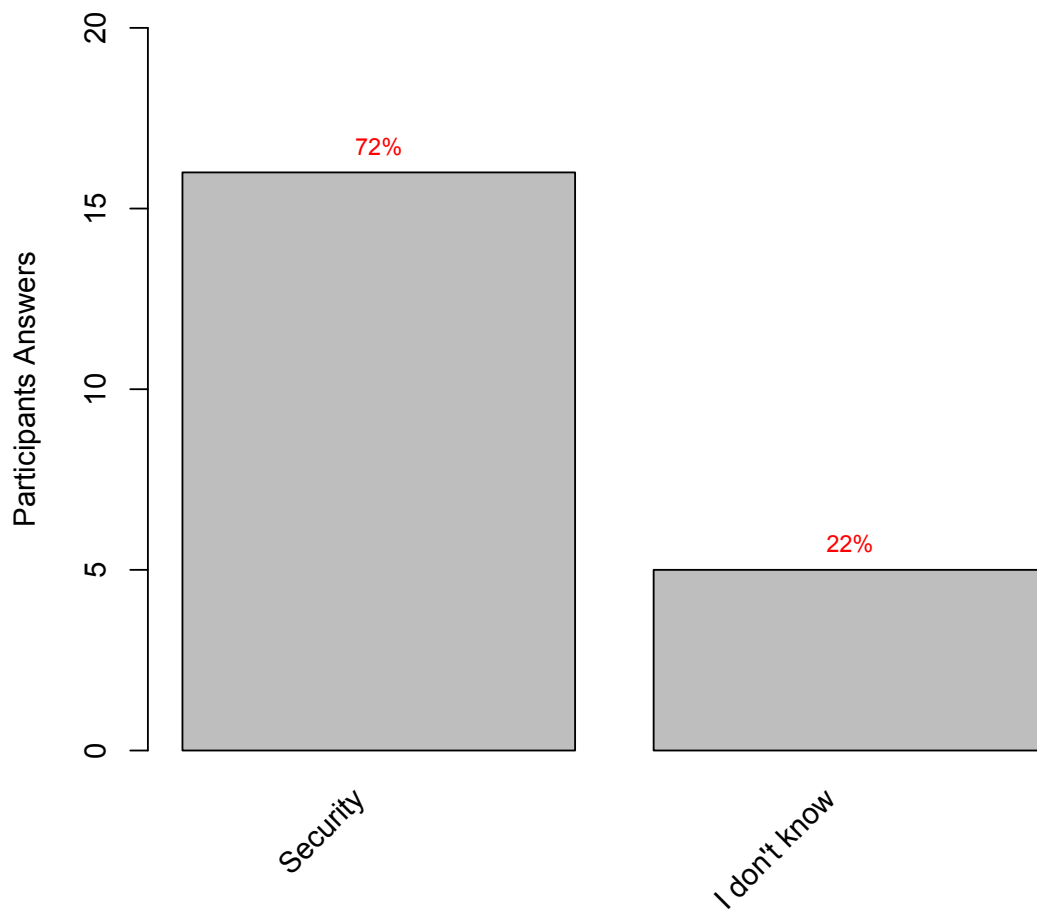


Figure 49: Perceived Implications of having unaligned Network Access settings

- Technical factors which means the underlying complexity of the task itself.
- Psychological factors including poor mental models or difficulty with risk assessment.
- Real-world factors comprising lack of financial incentives and production pressures

While our results similarly highlight number of real-world constraints, we identify that a key inhibitor toward secure software development practices is a “ it’s not my

responsibility” attitude.

Hybrid platforms have improved over time. Current distributions have a content security policy (CSP) to restrict local access per page. Developers are required to understand CSP meaning and learn how to customize it when needed. We have asked the developers if they know what is CSP or what it stands for. Only *one* participant was able to recognize it’s meaning.

Then, we asked if they would normally check the configuration of the app. Only *one* developer indicated that she would check the configuration file. All other developers indicated that they don’t normally check the file for many reasons including lack of time, lack of understanding, security is not a priority, and for many of them, it is not necessary as long as the app is working properly.

7.4.5 Q2: Perception of the benefits of CORDOVACONFIG

All participants indicated that they are likely to use the tool to configure their apps. One indicated that this tool is most beneficial when prototyping apps. By asking about the benefit/value of this tool, we show answers distribution in Figure 50. Easiness is the most observed benefit of the tool, depicted by answers such as:

[P7] “It is very tough to code hybrid apps, there is plenty of stuff to search and look for. Having this would make it less hectic”.

[P3] “Managing permissions is easy now”.

Many participants indicated that using this tool would enhance the security of their

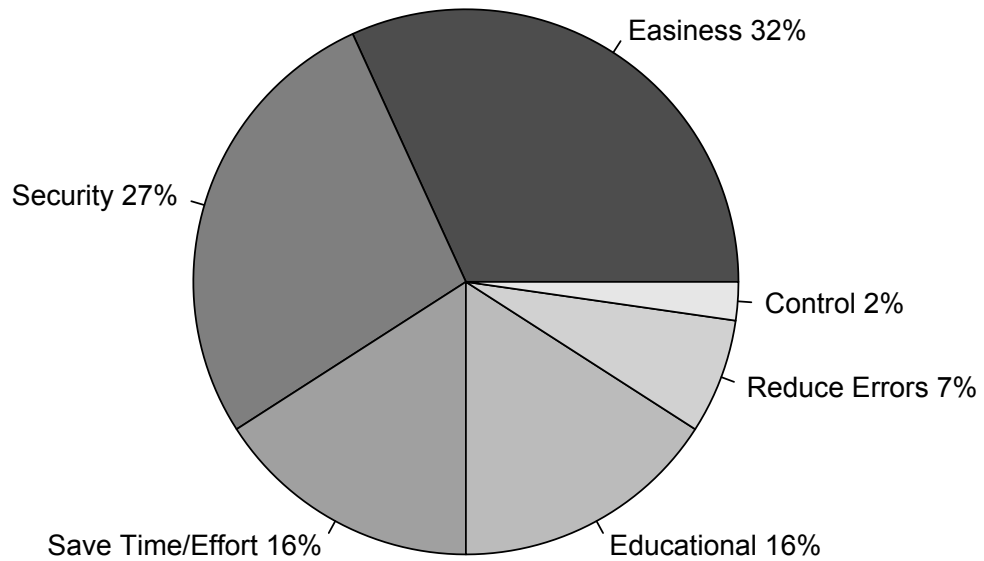


Figure 50: Perceived Benefits of CORDOVACONFIG

apps as well, example answers include:

[P1] “It restrict access to URLs, installed apps and plugins”.

[P4] “It automates the configuration according to app behavior so that your app is secure”.

Many participants indicate that this tool has increased their understanding of hybrid platform configurations in general and the app configurations in specific. Examples of participants responses include:

[P13] “It helped understand the app flow”.

[P6] “It enhances readability of Cordova configurations”.

[P16] “I was not aware of these settings and what they do. I am more aware now and I feel I have more control”.

[P19] “I appreciate all descriptions and explanations since we programmers tend to shorthand things easily”.

A few participants indicated that using this tool gave them more control over the app. Example comment:

[P13] “It seems like it is a good tool to insure that the app do what is supposed to do”.

Only two participants described generating page-wise level settings as a value of this tool, such as:

[P10] “Sometimes we don’t need plugins in certain pages, this generates configurations per page”.

On the whole, participants showed a high level of excitement, interest, and gratitude while and after using the tool. This can be attributed to the change from the original configuration process or even to the idea of bringing this whole step to their attention.

7.4.6 Tool Usability

Although usability is generally a subjective feature, assessing the general quality of appropriateness of an artifact is still imperative. Consider how even the most effective tool created can only be as useful as its users consider it to be. The same can be said

for this configuration tool though quantitatively proven to be effective so far, if its implementation is not considered usable by users, then it is still rendered an ineffective approach. As a result, we used the industry-standard System Usability Scale (SUS) to evaluate the usability of our CORDOVACONFIG based on the responses to the provided Likert-scale questions. SUS yields a single number representing a composite measure of the overall usability of the system being studied. It is important to note that though it is tempting to interpret the score as a percentage, it is not such, nor is it meant to be diagnostic, but simply an evaluation of an applications ease of use [42]. Based on the formula described in [67], we generated an overall SUS score of 86.25 for the tool. According to prior research, a SUS score above 68 is above average for general applications. Figure 51 represents the SUS scale for general applications, with a marker indicating how the CORDOVACONFIG tool's score compares. Figure. 52

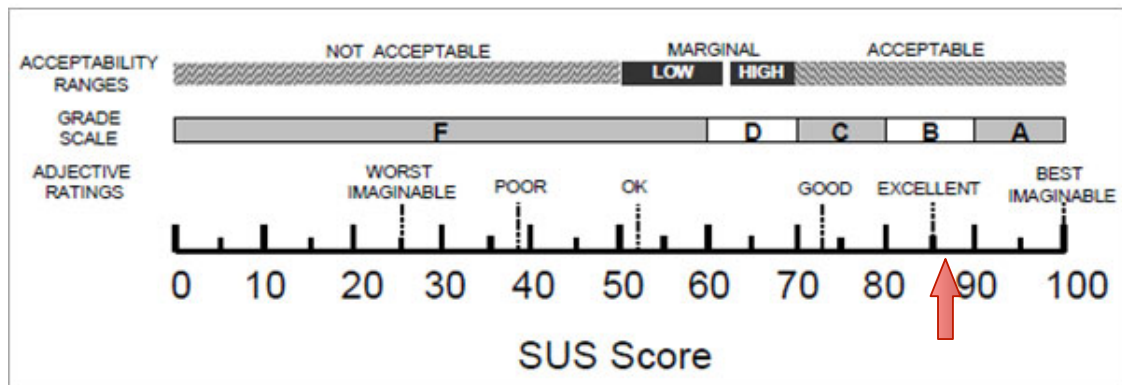


Figure 51: SUS Scale with red Arrow indicating CORDOVACONFIG Score

shows a descriptive statistics of SUS.

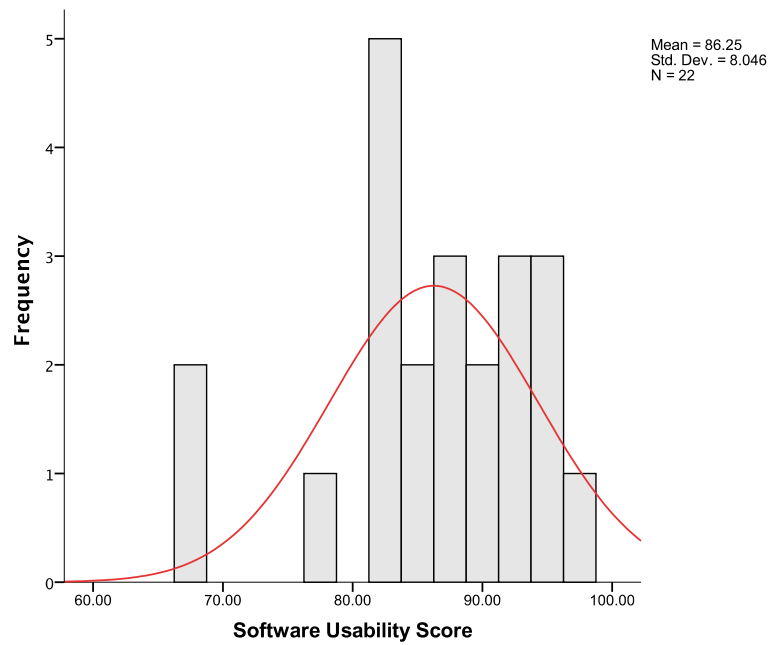


Figure 52: SUS Distribution

7.5 Limitations

Considering the novelty of the platform, conducting this user study have faced the following obstacles:

- Finding participants who have the required background needed to conduct the user study.
- Most of the participants are novice programmers in hybrid mobile apps in general and cordova-based apps in specific.

CHAPTER 8: CONCLUSION

Considering the rapid growth in the use of hybrid mobile apps, researchers must consider establishing a security framework by integrating models in the middleware itself rather than using additional layers of security. In an effort to transition from being *reactive* to *proactive* in securing hybrid apps, this work has addressed the importance of having more fine-grained access models, adopting more aligned configurations settings and incorporating automated tools to secure hybrid apps through recommended configurations. Towards this goal, we have designed, implemented, and conducted user studies to configure Cordova-based apps including more aligned and fine-grained configurations that conforms with the apps' behavior. Our design adds no effort on developers, and still generates policies to control app behavior at runtime. The policies aim to control plugin access at the state level and also controls apps behavior in terms of state transitions. Most of the implementation of this tool is done on the native side which is more stable and centralized. We developed a working prototype of our tool and used it for configuring representative applications to demonstrate the viability of the tool and its applicability to real-world scenarios. We have also measured the overhead time of using this tool compared to the stock version of the library, and results showed that the added overhead is minimal.

The main security limitation of having coarse-grained access model has been discussed already by related literature. However, to the best of our knowledge, the work

we present in this dissertation is the only work that addresses these limitations by changing the configuration scheme. As mentioned earlier, we regard configurations as the first line of defense as many attacks can be simply neutralized by proper app configurations. This work also carries the novelty of presenting tooling support that caters to developers of mobile hybrid apps. Our goal is not only to have fine-grained and more comprehensive configuration model, but also to provide support to developers in the form of an automated tool.

We realize that protecting apps through configuration is a vital need, but we also recognize that configurations alone do not guarantee full protection against potential attacks. Extension of this work should propose a pro-active plugin to help developers follow recommended secure coding while in the development stage. Most security breaches are a result of poor coding practices and improper data validation. More research work should address this need and propose solutions to harden the security of the code.

More research is also needed in the direction of addressing the security holes that may result from the new software stack of this specific category. More specifically, further analysis of the applicability of both web-based attacks and native code based attacks on hybrid apps. This is important because such attacks can be easily launched and with the given bridge provided by the library, the effect is amplified. One example is the new discussion about return-oriented attacks and their applicability on hybrid apps.

There is also an essential need to continue the line of exploratory research that aims to measure the effectiveness of current security measures and the trend of using

them among developers. This should give the platform vendors a better idea on how to re-design the platform and to how to address the current needs.

A current issue with hybrid apps security research is that most research body discusses detection of security issues such as data leakage through means of static and dynamic code analysis. While this is essential, but it is on the reactive side rather than the proactive one. Once the damage happens, it is hard to revert it. More research should present proactive solutions either in the middleware level or application level. It is the turn now to focus more on facilitating the consideration and the implementation of security principles as early as possible.

Finally, as mobile apps continue to dominate and as hybrid or cross-platforms in specific are expected to trump conventional development approaches, there is a basic need to further the research and efforts to make this approach more secure and usable. This work is a step towards better understanding and improving the security of mobile hybrid apps.

REFERENCES

- [1] Android api guide [permission]. <https://developer.android.com/guide/topics/manifest/permission-element.html>.
- [2] Android normal permissions. <https://developer.android.com/guide/topics/permissions/normal-permissions.html>.
- [3] Apache cordova. <https://cordova.apache.org>.
- [4] Cordova config. https://cordova.apache.org/docs/en/latest/config_ref/index.html/.
- [5] Domain whitelist guide. <https://cordova.apache.org/docs/en/1.8.0/guide/whitelist/index.html>.
- [6] Google chrome help forum. <https://productforums.google.com/forum/#!msg/chrome/yxX-9Fn8Hq8/UDXaH1PCDwAJ>.
- [7] Html5 security cheat sheet. https://www.owasp.org/index.php/HTML5_Security_Cheat_Sheet.
- [8] Phonegap platform security. <https://github.com/phonegap/phonegap/wiki/Platform-Security>.
- [9] Security guide. <https://cordova.apache.org/docs/en/3.5.0/guide/appdev/security/index.html>.
- [10] What is gstatic? (remove gstatic virus) (july 2017 update). <https://howtoremove.guide/what-is-gstatic-remove-gstatic-virus/>.
- [11] Whitelist documentation. <https://github.com/apache/cordova-plugin-whitelist>.
- [12] Wiki-domains: gstatic.com. <https://en.wiki-domains.net/wiki/gstatic.com>.
- [13] S. Amatya and A. Kurti. Cross-platform mobile development: challenges and opportunities. In *ICT Innovations 2013*, pages 219–229. Springer, 2014.
- [14] Android. WebView. "<http://developer.android.com/reference/android/webkit/WebView.html>", 2014.
- [15] AppBrain. Android statistics /cordova. <http://www.appbrain.com/stats/libraries/details/phonegap/phonegap-apache-cordova>.
- [16] R. Balebako and L. Cranor. Improving app privacy: Nudging app developers to protect user privacy. *IEEE Security & Privacy*, 12(4):55–58, 2014.

- [17] A. B. Bhavani. Cross-site scripting attacks on android webview. *CoRR*, abs/1304.7451, 2013.
- [18] A. Blyth. Secure coding principles and practices. *Infosecurity Today*, 1(3):46, 2004.
- [19] A. D. Brucker and M. Herzberg. On the static analysis of hybrid mobile apps. In *International Symposium on Engineering Secure Software and Systems*, pages 72–88. Springer, 2016.
- [20] A. D. Brucker and M. Herzberg. On the static analysis of hybrid mobile apps: A report on the state of apache cordova nation. In J. Caballero and E. Bodden, editors, *International Symposium on Engineering Secure Software and Systems (ESSoS)*, Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, 2016.
- [21] O. (c). Owasp top 10. <https://www.owasp.org/index.php/Top10#tab=Main>.
- [22] Y.-L. Chen, H.-M. Lee, A. B. Jeng, and T.-E. Wei. Droidcia: A novel detection method of code injection attacks on html5-based mobile apps. In *Trustcom/Big-DataSE/ISPA, 2015 IEEE*, volume 1, pages 1014–1021. IEEE, 2015.
- [23] E. Chin, A. P. Felt, V. Sekar, and D. Wagner. Measuring user confidence in smartphone security and privacy. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, page 1. ACM, 2012.
- [24] 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.
- [25] C. Coenraets. Tutorial: Developing a phonegap application. <http://coenraets.org/blog/phonegap-tutorial/>.
- [26] E. D. Corporation. Mobile Developer Population Reaches 12M Worldwide, Expected to Top 14M by 2020. <https://evansdata.com/press/viewRelease.php?pressID=244>.
- [27] M. Corporation. Cve details. <https://www.cvedetails.com>.
- [28] M. Corporation. Cve details for cordova. https://www.cvedetails.com/vulnerability-list/vendor_id-45/product_id-27153/Apache-Cordova.html.
- [29] I. Dalmaso, S. K. Datta, C. Bonnet, and N. Nikaiein. Survey, comparison and evaluation of cross platform mobile application development tools. In *2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 323–328. IEEE, 2013.
- [30] G. Dini, F. Martinelli, A. Saracino, and D. Sgandurra. Probabilistic contract compliance for mobile applications. In *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, pages 599–606. IEEE, 2013.

- [31] N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan. Security-by-contract: Toward a semantics for digital signatures on mobile code. In *European Public Key Infrastructure Workshop*, pages 297–312. Springer, 2007.
- [32] M. Georgiev, S. Jana, and V. Shmatikov. Breaking and fixing origin-based access control in hybrid web/mobile application frameworks. In *NDSS symposium*, volume 2014, page 1. NIH Public Access, 2014.
- [33] Google. Google Play. "<https://play.google.com/store/apps>", August 2013.
- [34] M. Green and M. Smith. Developers are not the enemy!: The need for usable security apis. *IEEE Security & Privacy*, 14(5):40–46, 2016.
- [35] T. Groß and T. Müller. Protecting javascript apps from code analysis. In *Proceedings of the 4th Workshop on Security in Highly Connected IT Systems*, pages 1–6. ACM, 2017.
- [36] M. L. Hale and S. Hanson. A testbed and process for analyzing attack vectors and vulnerabilities in hybrid mobile apps connected to restful web services. In *Services (SERVICES), 2015 IEEE World Congress on*, pages 181–188. IEEE, 2015.
- [37] B. Hassanshahi, Y. Jia, R. H. Yap, P. Saxena, and Z. Liang. Web-to-application injection attacks on android: Characterization and detection. In *Computer Security—ESORICS 2015*.
- [38] P. Inc. PhoneGap Inc. "<http://www.phonegap.com/>", 2013.
- [39] S. Institute. 2016 state of application security: Skills, configurations and components. <https://www.sans.org/reading-room/whitepapers/analyst/2016-state-application-security-skills-configurations-components-36917>.
- [40] D. Jaramillo, V. Ugave, R. Smart, and S. Pasricha. Secure cross-platform hybrid mobile enterprise voice agent. In *Southeastcon 2014, Ieee*, pages 1–6. IEEE, 2014.
- [41] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri. 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*, pages 66–77. ACM, 2014.
- [42] X. Jin, T. Luo, D. G. Tsui, and W. Du. Code injection attacks on html5-based mobile apps. *arXiv preprint arXiv:1410.7756*, 2014.
- [43] X. Jin, L. Wang, T. Luo, and W. Du. Fine-grained access control for html5-based mobile applications in android. In *Proceedings of the 16th Information Security Conference (ISC)*. Citeseer, 2013.
- [44] M. E. Joorabchi, A. Mesbah, and P. Kruchten. Real challenges in mobile app development. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 15–24. IEEE, 2013.

- [45] A. J. Ko and B. A. Myers. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing*, 16(1):41–84, 2005.
- [46] N. Kudo, T. Yamauchi, and T. H. Austin. Access control mechanism to mitigate cordova plugin attacks in hybrid applications. *Journal of Information Processing*, 26:396–405, 2018.
- [47] S. Lee, J. Dolby, and S. Ryu. Hybridroid: static analysis framework for android hybrid applications. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*, pages 250–261. IEEE, 2016.
- [48] 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.
- [49] 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.
- [50] J. Madden. Why html5 apps are ideal for enterprise mobility. <http://searchmobilecomputing.techtarget.com/feature/Why-HTML5-apps-are-ideal-for-enterprise-mobility>.
- [51] B. A. Myers and J. Stylos. Improving api usability. *Commun. ACM*, 59(6):62–69, May 2016.
- [52] M. Palmieri, I. Singh, and A. Cicchetti. Comparison of cross-platform mobile development tools. In *Intelligence in Next Generation Networks (ICIN), 2012 16th International Conference on*, pages 179–186. IEEE, 2012.
- [53] I. S. T. L. W. Paper. Ibm, native, web or hybrid mobile app development. 2012.
- [54] P. H. Phung, A. Mohanty, R. Rachapalli, and M. Sridhar. Hybridguard: A principal-based permission and fine-grained policy enforcement framework for web-based mobile applications.
- [55] T. Progress. The state of hybrid mobile development. <http://developer.telerik.com/featured/the-state-of-hybrid-mobile-development/>.
- [56] F. Raja, K. Hawkey, and K. Beznosov. Revealing hidden context: improving mental models of personal firewall users. In *Proceedings of the 5th Symposium on Usable Privacy and Security*, page 1. ACM, 2009.
- [57] C. Rizzo, L. Cavallaro, and J. Kinder. Babelview: Evaluating the impact of code injection attacks in mobile webviews. *arXiv preprint arXiv:1709.05690*, 2017.
- [58] J. R. Ruthruff, S. Prabhakararao, J. Reichwein, C. Cook, E. Creswick, and M. Burnett. Interactive, visual fault localization support for end-user programmers. *Journal of Visual Languages & Computing*, 16(1):3–40, 2005.

- [59] i. salesforce.com. Native, html5, or hybrid: Understanding your mobile application development options. https://developer.salesforce.com/page/Native,_HTML5,_or_Hybrid:_Understanding_Your_Mobile_Application_Development_Options.
- [60] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.
- [61] M. Shehab and A. AlJarrah. Reducing attack surface on cordova-based hybrid mobile apps. In *Proceedings of the 2nd International Workshop on Mobile Development Lifecycle*, pages 1–8. ACM, 2014.
- [62] K. Singh. Practical context-aware permission control for hybrid mobile applications. In *Research in Attacks, Intrusions, and Defenses*, pages 307–327. Springer, 2013.
- [63] K. Singh and J. Buford. Developing webrtc-based team apps with a cross-platform mobile framework. 2016.
- [64] D. Sun, C. Guo, D. Zhu, and W. Feng. Secure hybridapp: A detection method on the risk of privacy leakage in html5 hybrid applications based on dynamic taint tracking. In *Computer and Communications (ICCC), 2016 2nd IEEE International Conference on*, pages 2771–2775. IEEE, 2016.
- [65] G. Susan Moore. Gartner says demand for enterprise mobile apps will outstrip available development capacity five to one. <http://www.gartner.com/newsroom/id/3076817>.
- [66] Symantic. Web attack: Malicious javascript redirection 2. http://www.symantec.com/security_response/attacksignatures/detail.jsp?asid=28341.
- [67] Usability.gov. System Usability Scale (SUS). <http://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html>.
- [68] W3C. Hash uris. <http://www.w3.org/blog/2011/05/hash-uris/>.
- [69] W3C. State chart xml (scxml): State machine notation for control abstraction. <https://www.w3.org/TR/scxml/>.
- [70] WIRED. Understanding technological hype cycles. <https://www.wired.com/2012/08/understanding-technological-hype-cycles/>.
- [71] 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.
- [72] J. Xie, B. Chu, and H. R. Lipford. Idea: interactive support for secure software development. In *International Symposium on Engineering Secure Software and Systems*, pages 248–255. Springer, 2011.

- [73] J. Xie, B. Chu, H. R. Lipford, and J. T. Melton. Aside: Ide support for web application security. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 267–276. ACM, 2011.
- [74] J. Xie, H. Lipford, and B.-T. Chu. Evaluating interactive support for secure programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2707–2716. ACM, 2012.
- [75] J. Xie, H. R. Lipford, and B. Chu. Why do programmers make security errors? In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*, pages 161–164. IEEE, 2011.
- [76] G. Yang, J. Huang, and G. Gu. Automated generation of event-oriented exploits in android hybrid apps. NDSS, 2018.
- [77] G. Yang, A. Mendoza, J. Zhang, and G. Gu. Precisely and scalably vetting javascript bridge in android hybrid apps. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 143–166. Springer, 2017.
- [78] L. Yang, X. Cui, C. Wang, S. Guo, and X. Xu. Risk analysis of exposed methods to javascript in hybrid apps. In *Trustcom/BigDataSE/I SPA, 2016 IEEE*, pages 458–464. IEEE, 2016.
- [79] J. Zhu, H. R. Lipford, and B. Chu. Interactive support for secure programming education. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 687–692. ACM, 2013.
- [80] M. F. Zibrán, F. Z. Eishita, and C. K. Roy. Useful, but usable? factors affecting the usability of apis. In *Reverse Engineering (WCRE), 2011 18th Working Conference on*, pages 151–155. IEEE, 2011.

APPENDIX A: CORDOVACONFIG Screen Shots

Figure 53: Start Screen

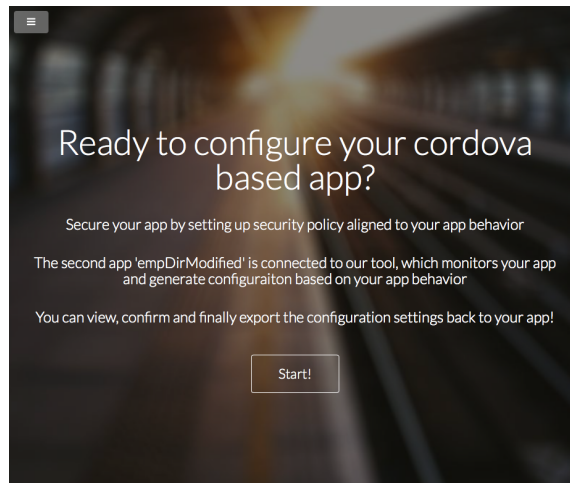


Figure 54: Current Configuration Analysis-Part1

Your App Configuration Analysis - Part 1

Plugins Found in your app config.xml

```

1 <feature name="Geolocation">
2   <param name="android-package" value="org.apache.cordova.geolocation.Geolocation" />
3 </feature>
4 <feature name="Camera">
5   <param name="android-package" value="org.apache.cordova.camera.CameraLauncher" />
6 </feature>
7 <feature name="Contacts">
8   <param name="android-package" value="org.apache.cordova.contacts.ContactManager" />
9 </feature>
10 <feature name="Capture">
11   <param name="android-package" value="org.apache.cordova.mediacapture.Capture" />
12 </feature>
13 <feature name="Battery">
14   <param name="android-package" value="org.apache.cordova.BatteryListener" />
15 </feature>

```

Which means accessing:

- Geolocation
- Video/ Audio capture
- Camera
- Contacts

[Next](#)

Figure 55: Current Configuration Analysis-Part2

Your App Configuration Analysis - Part 2

URLs the app is allowed to ask the system to open:

```
1 <allow-intent href="market:*" />
2 <allow-intent href="http://**/*" />
3 <allow-intent href="https://**/*" />
```

Which means accessing:

- Any Android Market
- Allow links to any **http** page to open in a browser
- Allow links to any **https** page to open in a browser

External apps the app is allowed to ask the system to open:

```
1 <allow-intent href="tel:*" />
2 <allow-intent href="sms:*" />
3 <allow-intent href="mailto:*" />
4 <allow-intent href="geo:*" />
```

Which means accessing:

- Allow to open the **dialler**
- Allow to open the **messages app**
- Allow to open the **email app**
- Allow to open the **maps**

[Next](#)

Figure 56: Current Configuration Analysis-Part3

Your App Configuration Analysis - Part 3

Network Requests (images, JS,etc) are allowed to be made:

```
1 <access origin="*" />
```

Which means accessing:

- Don't block any request

URLs the WebView can be navigated to:

No configuration found:

* By default, WebView is allowed to navigate to local files only

Android Permissions accesses are:

```
1 <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
2 <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
3 <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
4 <uses-permission android:name="android.permission.READ_CONTACTS" />
5 <uses-permission android:name="android.permission.WRITE_CONTACTS" />
6 <uses-permission android:name="android.permission.GET_ACCOUNTS" />
7 <uses-permission android:name="android.permission.RECORD_AUDIO" />
8 <uses-permission android:name="android.permission.RECORD_VIDEO" />
```

Which means accessing Android System:

- **Location** – An app can use the device's location. This includes two different location settings, approximate and precise. Using the approximate location option, the app gets the device's location from the network, and the precise option uses the device's GPS and network to determine the location. To do this, the permission allows the app to access extra location provider commands and GPS.
- **Photos/Media/Files** – The application has the ability to use the file on the device with the application installed. This includes reading and writing to the SD card and USB storage. The app can also mount and unmount external storage as well as format external storage. This permission deals with reading external storage on newer devices
- **Contacts**- find accounts on the device, see and modify the owner's contact card and add or remove contacts from the device
- **Video/Audio** - capturing and encoding a variety of common audio/ video formats integrated into your applications

[Next](#)

Figure 57: Plugin access captured for a state

App Behavior Confirm - Part 1:

We will walk you through different app behavior states along with resources accessed in each state
Start with you app life story as we have captured it, please confirm/ modify if needed

App State 1 - (S0):

index.html

Access the following plugin(s):

- Connection.

[Add More Plugins?](#)



Page(s) apply to this pattern require(s) the following permission(s):

- INTERNET.
- ACCESS_NETWORK_STATE.

Agree?

Figure 58: Plugin access captured for a state






App State 2 - (S1):

#employees/d+

Access the following plugin(s):

- Geolocation.
- Camera.
- Contacts.
- Connection.

[Add More Plugins?](#)

| Back | Details | Back | Details | Back | Details | Back | Details | Back | Details |
|---|--|---|--------------------------------------|---|--|---|--|---|---------------------------|
|  | Michael Scott Regional Manager |  | Pamela Beesly Receptionist |  | Meredith Palmer Supplier Relations |  | Phyllis Lapin Sales Representative |  | Ryan H Vice Pre |
| Call Office | 570-123-4567 | Call Office | 570-999-7874 | Call Office | 570-083-0107 | Call Office | 570-652-1919 | Call Office | 212-999-8887 |
| Call Cell | 570-865-2536 | Call Cell | 570-999-5555 | Call Cell | 570-588-6567 | Call Cell | 570-241-8585 | Call Cell | 212-999-8888 |
| SMS | 570-865-2536 | SMS | 570-999-5555 | SMS | 570-588-6567 | SMS | 570-241-8585 | SMS | 212-999-8888 |
| Add Location | Add Location | Add Location | Add Location | Add Location | Add Location | Add Location | Add Location | Add Location | Add Location |
| Add to Contacts | Add to Contacts | Add to Contacts | Add to Contacts | Add to Contacts | Add to Contacts | Add to Contacts | Add to Contacts | Add to Contacts | Add to Contacts |
| Change Picture | Change Picture | Change Picture | Change Picture | Change Picture | Change Picture | Change Picture | Change Picture | Change Picture | Change Picture |

Page(s) apply to this pattern require(s) the following permission(s):

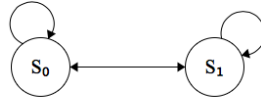
- ACCESS_COARSE_LOCATION.
- ACCESS_FINE_LOCATION.
- ACCESS_LOCATION_EXTRA_COMMANDS.
- WRITE_EXTERNAL_STORAGE.
- READ_CONTACTS.
- WRITE_CONTACTS.
- GET_ACCOUNTS.
- INTERNET.
- ACCESS_NETWORK_STATE.

Agree?

[Add More App State/ Plugin Accesses?](#)

Figure 59: App transition diagram

State Diagram of your app:



Next >>

Figure 60: App interaction with external components

App Behavior Confirm - Part2:

In this part, you will see the how your app interacted with external entities/ resources

Your app have accessed the following resources from external websites :

<http://liisp.uncc.edu/~yjaved/projects/img>

Add More External Access Origins?

Your app invoked these built-in apps? (check is you want to add more)

- System Messaging App
- System Dialer
- System Email
- System Maps App

Your app WebView navigated to these domains:

(*By default, Webview and iframes navigations only to file:// URLs. There is no non-http schemes allowed.)

Allow to navigate to external URLs?

Confirm Behavior

Figure 61: Generated Configurations

```

    <!-- config.xml -->
    <feature name="Geolocation" >
    <param name="android-package" value="org.apache.cordova.geolocation.GeoBroker" />
    <param name="state" value="#employees/\d+" />
    </feature>
    <feature name="Camera" >
    <param name="android-package" value="org.apache.cordova.camera.CameraLauncher" />
    <param name="state" value="#employees/\d+" />
    </feature>
    <feature name="Contacts" >
    <param name="android-package" value="org.apache.cordova.contacts.ContactManager" />
    <param name="state" value="#employees/\d+" />
    </feature>
    <feature name="Connection" >
    <param name="android-package" value="" />
    <param name="state" value="#employees/\d+" />
    </feature>
    <feature name="Connection" >
    <param name="android-package" value="" />
    <param name="state" value="index.html" />
    </feature>
    <access origin="http://liisp.uncc.edu/~yjaved/projects/img" />
    <allow-intent href="tel:*" />
    <allow-intent href="sms:*" />
    <!------->
    <!-- Add in config.xml to monitor app redirects -->
    <urlstate id="index" value="index.html" type=start location=local>
        <redirect id="#employeeed+">
        <redirect id="index">
    </urlstate>
    <urlstate id="#employeeed+" value="#employeeed+" location=local>
        <redirect id="#employeeed+">
        <redirect id="index">
    </urlstate>
    <!------->
    <!-- Android Manifest.xml -->
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
    <uses-permission android:name="android.permission.ACCESS_LOCATION_EXTRA_COMMANDS" />
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.READ_CONTACTS" />
    <uses-permission android:name="android.permission.WRITE_CONTACTS" />
    <uses-permission android:name="android.permission.GET_ACCOUNTS" />
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />

```

APPENDIX B: Survey Questions A & B

Background Questions

1. What is your education level
2. How many years you have been coding Mobile apps
3. How many years you have been coding web based applications
4. How many years have you been using Cordova/ PhoneGap platform
5. How many Cordova-Based apps you have developed?
6. How do you describe your knowledge in apps security (common attacks, best practices) in general ?
7. Check what describes a common coding practice you follow

(A) Pre-Survey Questions

Cordova Configuration Knowledge

1. Do you know what is config.xml and what is it for ? (One correct answer)
2. config.xml enables controlling: (Multiple answers)
3. How can you change app configuration ? (One correct answer)

Application Specific Configuration Knowledge

1. Check the plugins that are used in this app: (Multiple answers)

2. Check the Android permissions that are NOT needed in the app (Multiple answers)
3. According to current configuration, this app can load images from domain "http://liisp.uncc.edu" ? (One correct answers)
4. According to current configuration, Do you think you can use this href="data:image/jpeg;base64" (One correct answer)
5. According to current configuration, Do you think the app can open phone built-in Maps ? (One correct answer)

Developer Mental Model regarding granting permissions

1. Dialing a number (show a screen shot) is part of the application (One correct answer)
2. To include Messaging functionality in this app I need to (One correct answer)
3. This screen (List employees) requires the following permissions to run properly (One correct answer)

(A) Post-Survey Questions

Cordova Configuration Knowledge

1. config.xml is NOT controlling : (multiple correct answers)
2. How can you change app configuration ? (One correct answer)
3. Check the plugins that are NOT used in this app (multiple correct answers)

Application Specific Configuration Knowledge

1. Check the Android permissions that are needed in the app (Multiple correct answers)
2. According to current configuration, this app can execute remote javascript code from this server:"http://myserver.uncc.edu" (One correct answer)
3. The current app configuration allow this screen to appear(showing a browser screenshot): (One correct answer)
4. According to current configuration, Do you think the app can open phone built-in SMS ? (one correct answer)
5. Check the plugins that are NOT used in this app (Multiple correct answers)

Developer Mental Model regarding granting permissions

1. This screen (dialling a number) is part of my app (one correct choice)
2. To include this functionality (showing a screenshot of the built-in map app) in this app I need to (one correct answer)
3. This screen (saving a contact) requires the following permissions to run properly (One correct answer)

(B) Pre-Survey Questions

We have used the same questions used in the previous post-survey

(B) Post-Survey Questions

We have used the same questions used in the previous pre-survey APPENDIX C:
Interview Questions

1. Would you be interested to use this tool ? Why ? Why not ?
2. What is the value/benefit of such tool ?
3. What are the implications of keeping the default configurations ?
4. Can you think of an attack scenario that can compromise this app ?
5. What is the difference between old configurations and generated ones ?
6. Where the tool was confusing the most part ?
7. Do you know what is Content Security Policy?
8. Generally, would you care to go to the config file and check the configurations
? When would you do that ?