Scan Code Injection Flaws in HTML5-based Mobile Applications

Tuong Phi Lau
Faculty of Computer Engineering
University of Information Technology
Vietnam
laulpt@gmail.com

***Abstract*—HTML5-based mobile apps are becoming popular in the development of a cross-platform mobile. They are also built using web technologies, including HTML5, CSS, and JavaScript, so it may face with code injection attacks like web apps. However, code injection attacks are exploited in web apps and in mobile web apps are distinguished. The code injection attacks in web apps are often exploited by attackers throughout the cross-site scripting. In HTML5-based mobile apps, attackers can deploy attacks by various code injection channels such as inter-apps, inter-components, inter-devices communication, and local device resources such as WiFi, SMS, Contact. The plugin APIs are implemented for the code injection channels are defined as the sensitive plugin APIs. The previous approaches aimed at modeling known sensitive plugin APIs, and applying the data flow analysis to detect sensitive information flows from such modeled sensitive plugin APIs to vulnerable APIs. However, their method can miss code injection flaws caused by unknown sensitive plugin APIs. Besides, analyzing information flows in JavaScript is challenging. We found that the previous approaches are not able to analyze various contexts of callback functions. In this paper, we developed a static analysis tool called SCANCIF to scan code injection flaws. SCANCIF identifies the sensitive plugin APIs based on code injection tags, and analyzes flows of information based on modeling contexts of callback functions passed in function calls. We evaluated our approach on a data set of 3,204 HTML5-based mobile apps, as a result, SCANCIF scanned 220 vulnerable apps. We manually reviewed them and found 4 new code injection channels.**

Keywords—code injection attacks, PhonegGap, HTML5-based mobile apps, data flow analysis

I. INTRODUCTION

The development of smart phones over the years has leading software developers to switch to develop mobile applications. Native mobile apps are built using the platform-selected language such as Java in Android, and SWIFT in iOS. Developers need to write different versions of apps to support target platforms. They have the benefits that they are effective in the performance and can use all features of mobile devices integrated into apps such as the camera, contact. Unfortunately, developing native apps is time-consuming and expensive, because developers need to write different versions to support for multiple platforms. HTML5-based mobile apps are built using web technologies such as HTML5, CSS, and JavaScript. They can be run on smart devices using modern web browsers, e.g. WebView of Android and UIWebView of iOS. Developers only need to write one version of apps that can supports for multiple platforms, in addition to porting such apps from one platform to another platform much easier than native apps [4], [10]. Unfortunately, the web technologies pose security risks because they allow code and data to be mixed together. A string contains data and JavaScript code embedded into HTML pages to be displayed on a web browser. Attackers can exploit this vulnerability to inject malicious code into input strings. There are a variety of attacks that can be exploited in web apps and mobile apps, but we mainly address code injection attacks in HTML5-based mobile apps.

Since web apps are run on web servers and mobile apps are run on smart phones, they have the fundamental differences, especially, in the way of exploiting code injection attacks. The attackers usually exploit such kind of attacks throughout the cross-site channel [3], [13], [15], [24], [25] in web apps. For HTML5-based mobile apps, adversaries can take advantage of communication channels such as inter apps, inter components, inter devices to inject malicious code. Additionally, attackers can inject malicious code into local device resources that Jin et al. [7], [8] presented such as Contact, SMS. For example, attackers can inject malicious JavaScript code into the body of an SMS message. When the malicious SMS message is displayed on HTML5-based mobile app at vulnerable APIs, the malicious JavaScript code can get executed on a browser.

The previous papers proposed automatic detection tools for code injection flaws in HTML5-based mobile apps [7], [18]. Jin et al. [7] presented sensitive plugin APIs implemented for code injection channels such as Contact, SMS, Camera. Subsequently, they developed an automatic tool to detect dangerous information flows from such sensitive plugin APIs to unsafe rendering APIs. The work [18] is also similar to [7] by presenting a new text box injection channel, and introducing an automatic approach called DroidCIA to scan code injection faults. DroidCIA applied the depth first search (DFS) algorithm to slice sensitive data flow from the text box injection channel to sensitive sinks. The previous approaches [7], [18] detected code injection faults based on modeling exactly known sensitive plugin APIs implemented for code injection channels.

0-7695-6432-1/18/$31.00 ©2018 IEEE
DOI 10.1109/ICSTW.2018.00032

Developers can write different plugin APIs, so their approaches can miss unknown sensitive plugin APIs. In addition, analyzing data flow in JavaScript is challenging. We manually analyzed source code of HTML5-based mobile apps in [7], and we found that the previous approaches [7], [18] were not able to detect various contexts of callback functions.

In this paper, we introduce an automated analysis tool called SCANCIF (SCAN Code Injection Flaws) to detect code injection flaws by scanning sensitive information flows from sensitive plugin APIs to vulnerable sinks. We define sensitive plugin APIs as code injection channels such as SMS, Contact [7]. The sensitive plugin APIs are integrated into the middleware framework and can be seen as sensitive sources that can be injected malicious code by attackers. First, SCANCIF identifies sensitive plugin APIs based on code injection tags that we manually selected from code injection channels in [7]. We picked up possible code injection tags, including verbs, nouns, synonyms and abbreviations to identify sensitive plugin APIs. For example, the contact of a device can have tags such as "*find*", "*edit*", that can be used for finding or editing someone's first name in the contact list of a mobile device. Moreover, we did not aim at proposing a new approach to data flow analysis to analyze sensitive information flows in JavaScript. Instead, we made a slicing algorithm to analyze various contexts of callback functions that are often used to handle results in the plugin APIs (it will be detailed in Section III).

SCANCIF collects the predefined vulnerable APIs that allow JavaScript code to get executed. For each vulnerable API, it slices sensitive information flows from the sensitive plugin APIs to the predefined vulnerable API. Our approach is not fully automated. Vulnerable apps are verified by the manual code review.

In the previous evaluations, Jin et al. [7] evaluated their approach on a large set of 15,510 HTML5-based mobile apps. They flagged 471 vulnerable apps, and there are 15,039 remaining apps undetermined. Since many apps already removed from the Goolge Play Store, we were able to download 136 vulnerable apps from those 471 flagged apps, and download a data set of 3,068 apps of 15,039 remaining undetermined. Therefore, our evaluation experienced on a set of 3,204 (3,068+136) apps and are divided into two parts. First, we ran SCANCIF to analyze 3,068 apps, and it scanned 84 vulnerable apps that contain code injection flaws. We manually reviewed 84 apps and found 4 new code injection channels. Second, we ran SCANCIF to analyze 136 vulnerable apps from the list of 474 vulnerable apps. As a result, SCANCIF scanned sensitive data flows from six various sensitive plugin APIs against the approach in [7] that was able to scan sensitive data flows from two different sensitive plugin APIs.

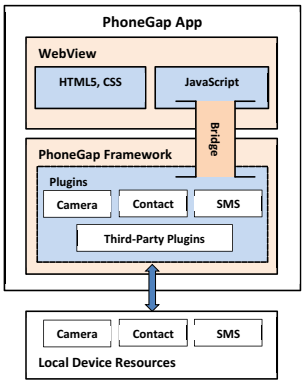

Fig. 1. Architecture of a PhoneGap app.

This paper makes the following academic contributions:

- We propose a solution to identify sensitive plugin APIs by categorizing code injection channels into code injection tags.
- We present information flow analysis based on modeling various contexts of callback functions.
- We present a statically automatic analysis tool called SCANCIF to scan code injection flaws in HTML5-based mobile apps.
- We evaluated SCANCIF on a data set of 3,204 HTML5-based mobile apps. Empirical results show that SCANCIF scanned 220 vulnerable apps.

The rest of the paper is organized as follows: Section 2 gives a brief overview of a PhoneGap app and information flow analysis. Section 3 describes our proposed approach. Section 4 shows evaluation results. Section 5 reviews related work, and section 6 summarizes the paper.

II. BACKGROUND

A. WebView and PhoneGap

WebView [29] is a powerful web browser in Android using HTML5 and CSS to display web contents, and JavaScript to make logic programming, but web contents usually come from untrusted and external sources. A sandbox mechanism is implemented inside WebView so as to prevent JavaScript code from untrusted sources executed on a web browser. However, this is not always the case for mobile web apps. WebView provides an API called *addJavascriptInterface()* as a communication bridge between JavaScript code and native code. The native code is used for accessing to local device resources in mobile device such as Contacts, file system, SMS, e.g. without the restriction of the sandbox mechanism of WebView. There are middleware frameworks that have been widely developed, including PhoneGap [2], RhoMobile [6], Appcelerator [19]. PhoneGap is the most popular, and it supports widely used mobile platforms, including Android, iOS, and Window Phone. Therefore, we describe it in our paper. Fig. 1 shows the architecture of a PhoneGap app.

PhoneGap is a middleware framework used for creating mobile web applications using web technologies such as HTML5, CSS, and JavaScript. Mobile apps developed by PhoneGap can be ported to multiple platforms easily. WebView provides JavaScript plugin APIs to call Java code plugin APIs inside the PhoneGap to communicate with the native system. By default, PhoneGap provides 16 plugin APIs such as Camera, Contact, file system that allow WebView to access to local device resources. However, developers can

```
1: class Contacts extends CordovaPlugin {
2:     boolean execute(String action, CordovaArgs args, CallbackContext
3:                                 callbackContext) {
4:         if (action.equals("find")) {
5:             String name = args.get(0).name;
6:             find(name, callbackContext);
7:         }
8:         else if (action.equals("save")) …
9:     }
10:    void find(String name, CallbackContext callbackContext) {
11:        Contact contact = query("SELECT list WHERE name=" + name);
12:        callbackContext.success(contact);
13:    }
14:    …..
```

Fig. 2. A Java plugin API.

```
1: var contacts = {
2:     find:function(fields, successCB, errorCB, options) {
3:         if (!fields.length) {
4:             errorCB();
5:         } else {
6:             var win = function(result) {
7:                 var cs = [];
8:                 for (var i = 0, l = result.length; i < l; i++)
9:                     cs.push(contacts.create(result[i]));
10:                successCB(cs);
11:            };
12:            exec(win, errorCB, "Contacts", "search", [fields, options]);
13:        }
14:    }
15:    ……
```

Fig. 3. A JavaScript plugin API.

write various plugin APIs to fulfill specification requirements of their apps. Fig. 3 shows the plugin API in JavaScript that uses *exec()* to call the plugin API in Java code in Figure 2.

B. *Code injection attacks in web apps and HTML5-based mobile apps*

Since web apps and HTML5-based mobile apps are developed by web technologies, they are subject to face with code injection attacks. However, they have fundamental differences in the way of exploiting attacks, specifically, in term of the attack channels that we describe in this study.

To deploy code injection or cross-site scripting attacks (XSS) in web apps, an attacker does not target a victim directly. Instead, attackers would exploit a vulnerability within a website or web application that the victim would visit, essentially using the vulnerable website as a vehicle to deliver a malicious script code to the victim's browser. While XSS can be taken advantage of within VBScript, ActiveX and Flash (although now considered legacy or even obsolete), unquestionably, the most widely abused is JavaScript because JavaScript is fundamental to most browsing experiences. Since attackers can only inject malicious code into the victim's browser via web servers, and there is no direct way for attackers to interact with victims, that is called the cross-site scripting channel or called the server channel.

HTML5-based mobile apps have more possible channels to exploit code injection attacks as in [7], [8] described. It can interact with many forms of entities such as its local system resources, including Wi-Fi, Contact, SMS, and communications such as inter-apps, inter-devices. Attackers can take advantage of such forms to exploit code injection attacks. For example, attackers can inject malicious JavaScript code into the body of an SMS message. When this message is displayed by unsafe rendering APIs on HTML5-based apps, the JavaScript code can get executed.

C. *Information flow analysis*

There are well-known frameworks implemented as dynamic and static analyzers such as Soot [32], TAJS [11], and WALA [33] to analyze vulnerabilities such as code injection flaws, private data leakages. The main challenges in analyzing those vulnerabilities are how to discover all sensitive flows, scale to large applications, and maintain the precision. Different types of apps such as Android, HTML5-based, and hybrid mobile apps have distinguished challenges in analyzing sensitive information flows from sensitive sources to vulnerable sinks because of their features. In Android apps, There are four main components, including activities are created by users, services perform background tasks, content providers define a database-like storage, and broadcast receivers listen for global events. The difficulty of analyzing sensitive information flows in these apps is modeling precisely and accurately Android model such as inter-components (between app components), inter-application (between separate installed apps), callback contexts, asynchronous calling components as [14], [16], [26] presented. In Hybrid apps, taking advantage of both native apps and web apps, hybrid apps become a promising solution. Because most hybrid apps are developed in multiple programming languages with different semantics, developers have to well understand such semantic differences. For example, Android hybrid apps are written in both JavaScript for user interaction and Java for accessing Android-specific features. Java code and JavaScript code interact with each other by foreign function calls. The difficulty in detecting sensitive information flows in hybrid apps not only modeling Android model as native apps, but also modeling communication bridges between the side of native Java code and the side of JavaScript code [25], [27], [28], [30], [31]. In HTML5-based apps, this kind of apps [7], [8], [9], [17], [28] is similar to hybrid apps. The challenges in identifying sensitive information flows in HTML5-based apps is how to identify sensitive plugin APIs implemented for code injection channels such as Contact, SMS, Call Log, Camera as described in Section 2.

III. APPROACH

We implemented a static analysis tool called SCANCIF to improve the following limitations. First, it can partially identify unknown sensitive plugin APIs based on code injection tags. Second, it can partially scan code injection fault with various sensitive information flows based on modeling the various contexts of callback functions.

A. *Identify sensitive plugin APIs*

We define the sensitive plugin APIs as sensitive sources that are integrated in the middleware framework such as Phonegap. We first classify code injection channels presented in [7], [8] into several categories, and we list possible code injection tags in TABLE I. We use the plural and singular verbs,

nouns, abbreviations and synonyms to make tags becoming diverse. For example, the plugin APIs for SMS can provide functions to read the title, body, or subject in a message. Also, the word "*body*" can have other synonyms likes "*content*", and "*message*" can have an abbreviation such as "*msg*".

TABLE I. CATEGORIES OF CODE INJECTION CHANNELS

Category	Tag
SMS	message, msg, content, title, subject, body, sms, rec, reception, receive
File	content, download, data, attach, entries, entry, path, dir, name, file, text, load, url
Contact	address, mobile, tel, phone, first, last, company, email, find, edit, delete, search, contact, update
Wi-Fi	accesspoint, wifi, ssid
Bluetooth	notification, device, dev, uuid
Media	meta, title, album, artist, format, picture, audio, video
Browser	cookies, history, bookmark
Other	bardcode, scan, nde, nfc, mime, intent, calendar, event

Besides, it also has other related tags to receive messages such as "*reception*", "*receive*". The tag "*scan*" is implemented in the plugin API (*barcodeScanner.scan*) in PhoneGap as the paper [7] presented, but there might have various sensitive plugin APIs that are implemented with a function, namely, "*scan"*. For instance, scanning information of devices in Bluetooth or Wi-Fi, scanning information from images of foods, and drinks in restaurants, e.g. Finally, we make a list of those code injection tags in TABLE I. Of course, we can add more categories and tags instead of using those listed as in TABLE I. We take an example at line 4 in Fig. 5, and line 7 in Fig. 7. readEntries is a sensitive plugin API, but it has different objects from two Fig. 5 and Fig. 7. Jin et al. [7] modeled 14 modeled sensitive plugin APIs, but only the plugin API, directoryReader.readEntries, with the object directoryReader is modeled. Thus, their tool missed the flow from the sensitive plugin API, Reader.readEntries in Figure 5, while our approach detects this code injection flaw.

Limitation. The problem of identifying sensitive plugin APIs based on code injection tags can cause false alarms. Actually, when writing code in JavaScript, many built-in and user-defined APIs are used in apps that can contain code injection tags as we listed in TABLE I, but those built-in and user-defined APIs are not sensitive plugin APIs. We manually made a list of safe function calls that are defined as not of the sensitive plugin APIs. Our detection tool bypasses those safe function calls based on that list during analysis. Unfortunately, it is impossible to make the list of all safe function calls because there are lots of great APIs available. Therefore, we made a program to slice the safe function calls that contain the code injection tags as in TABLE I, and also have callback functions from the most used APIs in JavaScript such as *AngularJS, jQuery Mobile, React*. This is due to the sensitive plugin APIs that usually have callback functions.

```
1: function clickScan() {
2:     window.plugins.barcodeScanner.scan(scannerSuccess, scannerFailure);}
3: function scannerSuccess(result) {
4:     if (result.cancelled == false) {
5:         $("#touch-to-scan").hide();
6:         $("#post-scan-container").show();
7:         var returnedText = result.text;
8:         var scannedFormat = result.format;
9:         resultSpan.innerHTML = returnedText; }}
```

Fig. 4. The first context of the sensitive information flow.

B. *Slicing information flow*

Jin et al. [7] applied static data flow analysis as an extension in WALA [33] to scan sensitive information flows from code injection channels to unsafe APIs in HTML5-based mobile apps. The work [18] introduced a detection tool extended from TAJS [11], a static analysis tool for JavaScript, to detect code injection flaws. However, TAJS and WALA's data flow analysis is insufficient to analyze various contexts of callback parameters. Our approach applied static analysis to analyze data flow based on the modeled contexts.

We conducted the manual code analysis by reviewing the source code of several vulnerable apps that Jin et al. [7] reported. We did not entirely review the source code of apps by hand. In fact, we wrote a program to slice vulnerable APIs of apps, then we manually reviewed information flows for those vulnerable APIs. As a result, we discovered two things. First, most of vulnerable apps in [7] have the same contexts of data flow from *barcodeScanner.scan* to vulnerable API. For example, the sensitive information flow comes from the variable, result, at line 3 to the vulnerable API, innerHTML, at line 9 in Figure 4. Second, we found that the previous approaches missed sensitive information flows in the following the contexts shown in Figure 5, 6, and 7. For instance, the sensitive plugin API, Reader.readEntries, at line 4 flows to the vulnerable API, append(), at line 16 as in Figure 5. Fig. 7 shows the sensitive plugin API is called to read entries, directoryReader.readEntries, at line 7, and the entry's name is displayed at line 15. Fig. 6 shows the meta data is collected by calling the plugin API, mediaFiles[i].getFormatData, at line 9, and those meta data are displayed at line 4, 5, 6.

```
1: function downloadComplete(x) {
2:     if (x == DOWNLOAD_COUNT){  //if the last one, trigger loadAssets
3:         var reader = DATADIR.createReader();
4:             reader.readEntries(function(entries) {
5:                 loadAssets(entries); },fail);
6:             return true; }
7:     return false; }
8: function loadAssets(entries) {
9:     var path = '';
10:         for (var i=0; i<entries.length; i++) {
11:             if(($.inArray(entries[i].name.split('/').pop(),knownfiles)===-1)&&
!entries[i].isDirectory) {
12:                 tmp = entries[i].name;
13:                 entries[i].remove(function(){console.log('unused asset removed:
'+tmp);},function(){console.log('could not remove asset: '+tmp);});}
14:             else if (!entries[i].isDirectory){
15:                 path = entries[i].toURL();
16:                 $('#body_content').append("path: " + path + "<br />");
```

Fig. 5. The second context of the sensitive information flow.

```
1: function captureVideoSuccess(mediaFiles) {
2:   var i, len;
3:   var formatSuccess = function (mediaFile) {
4:       $('#format-data').append("Height:  <strong>"  +  mediaFile.height  +
"</strong><br/>");
5:       $('#format-data').append("Width:  <strong>"  +  mediaFile.width  +
"</strong><br/>");
6:       $('#format-data').append("Duration: <strong>" + mediaFile.duration/1000 +
"s</strong><br/>");};
7:       for (i = 0, len = mediaFiles.length; i < len; i += 1) {
8:           $('#capture-result').html("<strong>" + (i+1) + " files</strong>");
9:           mediaFiles[i].getFormatData(formatSuccess, formatError);}
10:    console.log("captureMediaSuccess");}
11: function captureAudio() {
12:   $('#format-data').empty();
13:   $('#capture-result').empty();
14:   navigator.device.capture.captureAudio(captureAudioSuccess,  captureError,
{limit: 1});}
```

Fig. 6. The third context of the sensitive information flow.

Let's consider several information flows from the real examples.

The function call, mediaFiles[i].getFormatData, at line 9 passes the callback function, formatSuccess, in Figure 6. This callback function is defined as a variable that stored the function expression at line 3 in Figure 6. Take a look at Figure 5, the anonymous callback function at line 5 has the callsite, loadAssets(entries). From the observed examples, we made a slicing algorithm to analyze the data flow based on modeling various contexts of callback functions as follows.

+ The function calls have callback functions. This is due to that the real examples in Figure 4, 5, 6 show the majority of the sensitive plugin APIs of the middleware frameworks that are implemented with callback handlers. Also, we manually review the plugin APIs of Phonegap [2] and found that all plugin APIs are mostly implemented using callback functions to handle results.

+ The callback function of the function calls that is defined as a variable storing the functional expression, and that is passed as an anonymous function.

Limitation. Sensitive plugin APIs can be developed without callback functions, so the proposed approach can miss those sensitive flows. Also, callback functions are passed as parameters in function calls to handle results can be written in various ways, so it can miss those sensitive flows. In addition, there might have various contexts of information flows in apps. This method solves partially the problem of data flow analysis.

C. *SCANCIF Overview*

We implemented SCANCIF in two steps as follows.

Step 1: We configure some preconditions by making manually a list of code injection tags, a list of safe functions following Section III-A, and modeling the contexts of information flows following Section III-B.

Step 2: TAJS parses HTML files including JavaScript sources into a flow graph which each node is equivalent to an instruction, each block contains a sequence of consecutive

```
1: function totalRecordings() {
2:     window.requestFileSystem(LocalFileSystem.PERSISTENT, 0,
3:         function(fileSys) {
4:             fileSys.root.getDirectory(RecordFolder, {create: true,exclusive: false},
function(directory) {
5:                 directoryFullPath = directory.toURL();
6:                 var directoryReader = directory.createReader();
7:                 directoryReader.readEntries(successFoldersToRender,fail);
8:             }, fail);
9:         }, fail);
10: }
11: function successFoldersToRender(entries) {
12:     var i;
13:     for (i = 0; i <= entries.length; i++ ) {
14:         if (entries[i] === undefined) {return;}
15:         $('#contributionListRecorder').append('<li  class="audioHolder"  ><a>' +
entries[i].name +'</a></li>'); }
16: function fail(error){
17:     console.log("ERROR");
18:     console.log(JSON.stringify(error));
19:     $('#body_content').append('tripped error' + JSON.stringify(error));}
```

Fig. 7. The fourth context of the sensitive information flow.

nodes, and each edge represents the control flow between the instructions in the program. From the flow graph, it collects the predefined vulnerable APIs such as *append(), html(), innerHTML* which are commonly used in HTML5-based mobile apps as Jin et al. [7] shown. For each the predefined vulnerable API, it slices sensitive information flows from the sensitive plugin APIs that contains code injection tags, and are not of the list of the predefined safe functions to the predefined vulnerable API as in the algorithm 1. The information flow analysis *backwardslice(n)* is done based on the modeled contexts following Section III-B.

Algorithm 1 Scan code injection flaws

```
N:= a list of the predefined vulnerable APIs
T:= a list of the code injection tags
SF:= a list of the predefined safe functions
void analyze() {
    for n in N do
        sensitivePluginAPI = backwardslice(n)
        if (sensitivePluginAPI.contain(T) &&
                    sensitivePluginAPI != SF) then
            reportFlaws()
        end if
    end for
}
```

IV. EVALUATION

We implemented SCANCIF in about 1,000 lines of Java code as an extension of TAJS [11], an open source static analysis tool for JavaScript. Jin et al. [7] conducted intensively their evaluation on a large set of 15,510 HTML5-based mobile apps, as a result, they flagged 474 vulnerable apps, and there still have 15,039 remaining apps undetermined. In our evaluation, we conducted an experiment to evaluate a smaller set of data 3,204 from 15,510 apps, and compare SCANCIF with MCIFINDER [7], a static analysis tool for code injection flaws on HTML5-based mobile apps. We divided our experiments that applied the algorithm 1 into two parts. In the first experiment, we ran SCANCIF to analyze 136 vulnerable apps taken from 474 flagged apps as the previous study

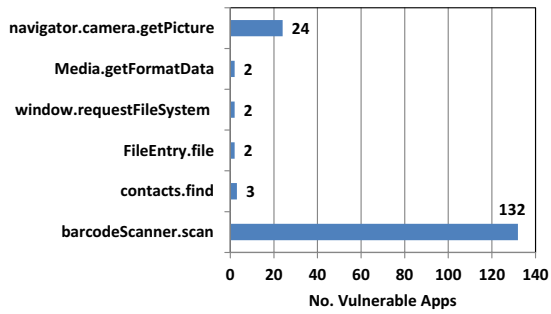

Fig. 8. Code injection channels identifed from 136 vulnerable apps.

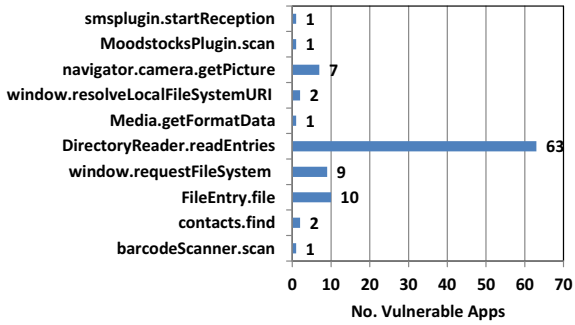

Fig. 9. Code injection channels identifed from a data set of 3,068 apps.

reported [7]. In the second experiment, we ran SCANCIF to analyze a data set of 3,068 of 15,039 remaining apps undetermined. We wrote a script file to run SCANCIF automatically for the experiments with total 3,204 (3,068+136) apps. We limited the processing time of each app within 30 seconds. Our hardware configuration is Intel®Core™ i7 2.6 GHz (4 cores), RAM 16GB.

A. Experiment on 136 vulnerable apps

Since some apps of 474 vulnerable apps in [7] reported were already removed from Google Play, we were able to download 136 vulnerable apps for evaluations. In the previous results, MCIFINDER [7] was able to scan the sensitive information flows from only two sensitive plugin APIs, namely, *barcodeScanner.scan* and *contacts.find* to vulnerable APIs. In our experiment, Fig. 8 shows that SCANCIF is able to scan the sensitive flows of information from six various sensitive plugin APIs to vulnerable APIs. Those six sensitive plugin APIs were confirmed as code injection channels following [7], [8].

B. Experiment on a dataset of 3,068 apps

We were able to download 3,068 apps from 15,039 remaining apps undetermined because many apps were already removed. In this experiment, SCANCIF scanned 84 vulnerable apps containing the code injection faults that have information flows from 10 sensitive plugin APIs to vulnerable APIs as in Figure 9, while MCIFINDER [7] was not able to scan those vulnerable apps. We manually reviewed 84 vulnerable apps, and found 4 new sensitive plugin APIs that will be described in the next part of this section. There have 63 out of 84 vulnerable apps containing the code injection flaws that have the sensitive plugin API with the API's name, "*readEntries"*. However, this

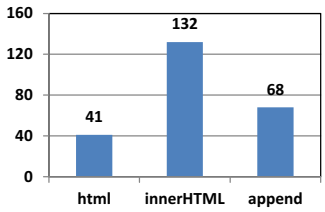

Fig. 10. Vulnerable APIs from 220 vulnerable apps.

API has the different object's name as the examples described in Section III-A. Therefore, we grouped the sensitive information flows from the sensitive plugin APIs with the API's name, "*readEntries"*, but with the different object's name, *DirectoryReader.ReadEntries, Reader.readEntries,* into the sensitive plugin API, *DirectoryReader.readEntries* as in Figure 9. Also, SCANCIF still can detect two vulnerable apps containing the code injection flaws that have the sensitive information flows from the plugin API *contacts.find*, and one vulnerable app that from the plugin API *barcodeScanner.scan*.

C. Sensitive plugin APIs

We evaluated our detection tool on a data set of 3,204 apps taken from a large set of 15,510 apps to compare with MCIFINDER [7]. As a result, SCANCIF scanned 220 vulnerable apps of 3,204 apps that contain code injection flaws. We manually reviewed and found 4 new sensitive plugin APIs. In addition, there have 132 vulnerable apps contain the sensitive flows of information from sensitive plugin APIs to the vulnerable API, *innerHTML*. Following with the vulnerable API, *append()*, with 68 vulnerable apps, and 41 vulnerable apps with *html()* in Figure 10. The vulnerable apps of four sensitive plugin APIs are shown in TABLE II, and four sensitive plugin APIs are described as follows.

navigator.camera.getPicture plugin API [20] is used for capturing or displaying a photo from the camera on a mobile device. We only considered the case of displaying the picture files on the mobile device. SCANCIF scanned 31 vulnerable apps that read the directory of camera pictures by calling this plugin API. Most of vulnerable apps display the directory of the pictures at vulnerable APIs. As the author [8] presented, the filename can be injected malicious code.

window.resolveLocalFileSystemURI [21] plugin API is used to access to the files in the system. SCANCIF scanned two vulnerable apps by calling this plugin API to read the directory of the filename, then the directory is displayed at vulnerable APIs. This case is similar to the previous plugin API.

MoodstocksPlugin.scan [22] plugin API is used to scan both the barcode and the images from the menu information in restaurants. Our detection tool detected one vulnerable app using this plugin API to scan the barcode and images from the

restaurant's menu to get food's name, drink's name, and these names are displayed at vulnerable APIs. Attackers can embed malicious code into the restaurant's menu. When victims scan information about foods, drinks, e.g. in menus, attackers can get full control to the victim's mobile device.

smsplugin.startReception [23] plugin API is used to receive incoming messages. SCANCIF scanned one vulnerable app using this API to receive messages, and those messages are displayed at vulnerable APIs in the app. Therefore, attackers can take advantage of this vulnerability to inject malicious code into those messages so as to trigger the code injection attack.

TABLE II. LIST OF VULNERABLE APPS OF 4 NEW CODE INJECTION

Apk name	Sensitve plugin APIs	Vulnerable APIs
MetroConstruction_v1.0_apkpure.com.apk	navigator.camera.getPicture	append
nl.brightwood.jankla assen_2013-06-30.apk	navigator.camera.getPicture	html
org.apache.cordova.o wnstagram_2013-01-15.apk	navigator.camera.getPicture	innerHTML
SmileyFace_v1.0.0_a pkpure.com.apk	navigator.camera.getPicture	append
org.openfoodfacts.sca nner_2014-12-11.apk	navigator.camera.getPicture	append
com.mindspacetech.g r8magazine_2014-04-30.apk	navigator.camera.getPicture	html
testbarcode.mediasoft .com_2011-08-24.apk	Navigator. camera. getPicture	innerHTML
org.openfoodfacts.sca nner_2014-12-11.apk	MoodstocksPlugin.scan	html
Beermadmobiletrial_ v1.17_apkpure.com.a pk	window.resolveLocalFileSyst emURI	html
com.CascadeAE.Mo bile.Android_2015-12-23.apk	window.resolveLocalFileSyst emURI	html
Boatsteward_v1.5_ap kpure.com.apk	smsplugin.startReception	innerHTML

D. Discussion

From our evaluation results, we make questions as follows.

1) If SCANCIF applies the slicing algorithm to analyze the information flows based on modeling contexts of callback functions as described in Section III-B, and identifies the sensitive plugin APIs based on modeling 14 sensitive plugin APIs following [7], [8], how many vulnerable apps?

a) Experiment on 136 vulnerable apps: SCANCIF still scanned 136 vulnerable apps, but those vulnerable apps contain the code injection faults with the sensitive information flows from five various sensitive plugin APIs instead of six various sensitive plugin APIs detected in Figure 8. It missed the code injection faults with the sensitive information flow from the sensitive plugin API, *navigator.camera.getPicture.*

b) Experiment on a data set of 3,068 apps: SCANCIF scanned only 27 vulnerable apps. We manually reviewed 27 apps, and found that it missed 4 vulnerable apps that contain sensitive information flows from 4 sensitive plugin APIs as described in the previous part of this section, and missed 53 vulnerable apps that contain sensitive flows from the sensitive plugin APIs, *reader.readEntries*, *dirReader.readEntries*. This is due to Jin et al. [7] considered *directoryReader.readEntries.*

2) If SCANCIF applies the slicing algorithm to analyze the sensitive information flows based on the modeled contexts of callback functions as described in Section III-B, and identifies the sensitive plugin APIs based on code injection tags as described in Section III-C, how many vulnerable apps?

In this case, SCANCIF applies the algorithm 1 in Section III-C, and the evaluation result is similar to the experiments conducted in the part A, B of this section.

V. RELATED WORK

Detect XSS vulnerabilities in web apps. Vogt et al. [12] introduced an approach using a dynamic taint analysis to prevent XSS attacks by tracking the flow of sensitive information in the web browser. Saxena et al. [15] proposed a hybrid approach called *taint enhanced black-box fuzzing* combining dynamic taint analysis with automated random fuzzing to detect vulnerabilities in web apps. Sebastian et al. [13] proposed a fully automated system to identify DOM-based XSS issues by integrating the vulnerability detection directly into the web browser's execution environment. However, XSS attacks between web apps and HTML5-based mobile apps are different as discussed in the background. Our work aims at detecting code injection flaws in HTM5-based mobile apps.

Detect vulnerabilities in Android apps. There are other researches focusing on detecting privacy leaks in Android apps. DroidSafe [14] is a static information flow analysis tool that analyzes potential leaks of sensitive information in Android applications. DroidSafe combines a comprehensive, accurate, and precise model of the Android runtime with static analysis decisions that enable the DroidSafe analyses to scale to analyze this model. Yang et al. [16] developed the AppIntent framework applying new symbolic execution technique to detect unintended data transmission. Fu et al. [17] designed the LeakSemantic framework that combines program analysis with machine learning to detect abnormal sensitive network transmission. In our work, we aim at detecting code injection faults that allow malicious code can get executed in HTML5-based mobile apps.

Detect vulnerabilities in HTML5-based mobile apps. The previous work [1], [5], [7], [8], [9], [18] aimed at detecting code injection flaws in HTML5-based mobile apps. Mao et al. [9] proposed an approach to detect injected behaviors in apps at run time. This approach tracks the execution of apps, and generates behavior state machines to describe the runtime behaviors of apps based on the execution contexts of apps. Once code injection happens, the injected behaviors will be detected based on deviation from the behavior state machine of the original app. Jin et al. [5] and Georgiev et al. [1] aimed to improve the access control mechanism to prevent attacks. Also, Jin et al [7], [8] described known code injection channels as sensitive sources and developed a detection tool to scan code injection faults. However, their approach can miss code

injection flaws causing by unknown sensitive plugin APIs. DroidCIA [18] was proposed to detect code injection faults, but it also identifies sensitive sources based on the predefined code injection channel as [7]. Moreover, their data flow analysis [7], [18] is not able to detect various contexts of sensitive information flows as presented in Section III-B. Our work is similar to detect code injection flaws as [7], [18], but we aimed at improving their limitations. First, our tool can identify unknown sensitive plugin APIs implemented for code injection channels as shown in Section III-A. Second, SCANCIF analyzes data flow based on modeling various contexts of callback functions as shown in Section III-B.

VI. CONCLUSION

In this paper, we introduce a tool to detect code injection flaws by scanning sensitive information flows from sensitive plugin APIs to vulnerable APIs in HTML5-based mobile apps. Our approach is not fully automated. The verification of code injection faults in vulnerable apps is executed by human analysis. SCANCIF identifies sensitive plugin APIs based on code injection tags. We manually selected the code injection tags from code injection channels that the previous work reported. In addition, SCANCIF is able to analyze various contexts of callback functions are passed in function calls such as anonymous callback functions, variables storing callback functions. We evaluated our detection tool on a data set of 3,204 HTML5-based mobile apps downloaded from Google Play Store. Experimental results show that SCANCIF scanned 220 vulnerable apps. We manually reviewed those vulnerable apps and found 4 new sensitive plugin APIs that can be injected code. All vulnerable apps are published at this link [34].

REFERENCES

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

[2] Phonegap: Build amazing mobile apps powered by open web tech. https://phonegap.com.

[3] HTML code injection and cross-site scripting: Understand the cause and effect of XSS. http://www.technicalinfo.net/papers/CSS.html.

[4] A. Charland, B. Leroux, "Mobile application development: web vs. native," In *Communications of the ACM*, 54(5), 2011, pp. 49-53.

[5] X. Jin, L. Wang, T. Luo, W. Du, "Fine-grained access control for HTML5-based mobile web applications in Android," In *Proceedings of 16th Information Security Conference (ISC)*, 2015, pp. 309-318.

[6] Rhomible suit. https://www.zebra.com/us/en/products/software/mobile-computers/rhomobile-suite.html.

[7] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, G. N. Peri, M. Young, "Code injection attacks on HTML5-based mobile apps: characterization, detection, mitigation," In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014, pp. 66-77.

[8] X. Jin, T. Luo, D. G. Tsui, W. Du, "Code injection attacks on HTML5-based mobile apps," In *arXiv preprint arXiv:1410.7756*, 2014.

[9] J. Mao, R. Wang, Y. Chen, Y. JIA, "Detecting injected behaviors in HTML5-based Android applications," In *Journal of High Speed Networks*, 2016, 22(1), pp. 15-34.

[10] N. Huy, D. Van, "Evaluation of mobile app paradigms," In *Proceedings of the 10th International Conference on Advances in Mobile Computing & Multimedia* (*MoMM'12)*, 2015.

[11] S. H. Jensen, A. Møller, P. Thiemann, "Type Analysis for JavaScript," In *Static Analysis Symposium (SAS)*, Vol. 9. 2009, pp. 238-255.

[12] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, G. Vigna, "Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis," In *Network and Distributed System Security* S*ymposium (NDSS),* 2007, p. 12.

[13] S. Lekies, B. Stock, M. Johns, "25 million flows later: large-scale detection of DOM-based XSS," In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2013, pp. 1193-1204.

[14] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, M. C. Rinard, "Information Flow Analysis of Android Applications in DroidSafe," In *Network and Distributed System Security Symposium (NDSS)*, 2015.

[15] P. Saxena, S. Hanna, P. Poosankam, D. Song, "FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications," In *Network and Distributed System Security Symposium (NDSS)*, 2010.

[16] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, X. S. Wang, "Appintent: Analyzing sensitive data transmission in android for privacy leakage detection," In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2013, pp. 1043-1054.

[17] H. Fu, Z. Zheng, S. Bose, M. Bishop, P. Mohapatra, "LeakSemantic: Identifying Abnormal Sensitive Network Transmissions in Mobile Applications," In *arXiv preprint arXiv:1702.01160*, 2017.

[18] Y. L. Chen, H. M. Lee, A. B. Jeng, T. E. Wei, "DroidCIA: A novel detection method of code injection attacks on html5-based mobile apps," In *Trustcom/BigDataSE/ISPA*, 2015.

[19] https://www.appcelerator.com/.

[20] https://cordova.apache.org/docs/en/latest/reference/cordova-plugin-camera/.

[21] https://cordova.apache.org/docs/en/2.0.0/cordova/file/localfilesystem/localfilesystem.html.

[22] https://github.com/stephanebachelier/moodstocks_phonegap_plugin_old

[23] https://github.com/asanka-x/Phonegap-SMS.

[24] S. Son, K. S. McKinley, V. Shmatikov, "Diglossia: Detecting code injection attacks with precision and efficiency," In *Proceedings of the ACM SIGSAC Conference on Computer & Communications Security*, 2013, pp. 1181-1192.

[25] L. K. Shar, H. B. K. Tan, L. C. Briand, "Mining SQL injection and cross site scripting vulnerabilities using hybrid program analysis," In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2013, pp. 642-651.

[26] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," In *ACM Sigplan Notices 49(6)*, 2014, pp. 259-269.

[27] S. Lee, J. Dolby, S. Ryu, "HybriDroid: Static analysis framework for Android hybrid applications," In *Automated Software Engineering (ASE), 31st IEEE/ACM International Conference* on 2016, pp. 250-261.

[28] M. Shehab, A. AlJarrah, "Reducing attack surface on Cordova-Based hybrid mobile Apps," In *Proceedings of the 2nd International Workshop on Mobile Development Lifecycle*, 2014, pp. 1-8.

[29] T. Luo, H. Hao, W. Du, Y. Wang, H. Yin, "Attacks on WebView in the android system," In *Proceedings of the 27th Annual Computer Security Applications Conference*, 2011, pp. 343-352.

[30] A. D. Brucker, M. Herzberg, "On the static analysis of hybrid mobile apps," In *International Symposium on Engineering Secure Software and Systems*, 2016, pp. 72-88.

[31] S. Amatya, A. Kurti, "Cross-platform mobile development: challenges and opportunities," In *ICT Innovations,* 2013, pp. 219-229.

[32] https://github.com/Sable/soot.

[33] https://github.com/wala/WALA.

[34] https://github.com/phituong/SCANCIF.