DroidCIA: A Novel Detection Method of Code Injection Attacks on HTML5-based Mobile Apps

Yen-Lin Chen
National Taiwan University of Science and Technology
Taipei 106, Taiwan
Email: M10215051@mail.ntust.edu.tw

Hahn-Ming Lee
National Taiwan University of Science and Technology
Taipei 106, Taiwan
Academia Sinica,
Nankang, Taipei 11529, Taiwan
Email: hmlee@mail.ntust.edu.tw

Albert B. Jeng
National Taiwan University of Science and Technology
Taipei 106, Taiwan
Email: albertjeng@hotmail.com

Te-En Wei
National Taiwan University of Science and Technology
Taipei 106, Taiwan
Email: d9807501@mail.ntust.edu.tw

***Abstract*—Smartphones have become more and more popular recently. There are many different smartphone systems, such as Android, iOS, etc. Based on HTML5, now developers can have a convenient framework to develop cross-platform HTML5-based mobile apps. Unfortunately, HTML5-based apps are also susceptible to cross-site scripting attacks like most web applications. Attackers can inject malicious scripts from many different injection channels. In this paper, we propose a new way to detect a known malicious script injected by using HTML5 text box input type along with "document.getElementById("TagID").value". This new text box injection channel was not detected by other researchers so far because they only analyzed JavaScript APIs, but overlooked HTML files which captured text box input type information. Later, we applied this new method to a vulnerable app set with 8303 cases obtained from Google Play. We detected a total of 351 vulnerable apps with accuracy 99%, which included 347 detected also by other researchers as well as 4 extra vulnerable apps that belonged to this text box injection channel. We also implemented a Code Injection Attack detection tool named *DroidCIA* that automated the drawing of JavaScript API call graph and the combination of API with HTML information.**

I. Introduction

The IDC report showed that smartphone shipments increased 25.2% from 2013 to 2014 [1]. Vendors shipped a total of 327.6 million units in 2014 [1]. The smartphone related app usage also increased considerably. There are three main smartphone operating systems, Android, iOS and Microsoft. Thus, it required the developers to develop three different mobile apps to run on these three systems respectively. Using a hybrid application framework allows the developers to develop mobile apps for various operating systems more conveniently because it could support cross-platform mobile app development. The report showed in [2] that 75% of developers use HTML5 for mobile app development . There are many hybrid application frameworks and the most popular one is PhoneGap [3].

Although HTML5-based mobile apps are portable across different mobile platforms, however, they are still susceptible to cross-site scripting attacks (XSS) like most web applications [4]. For example, an attacker could use many more channels to inject code than XSS [4]. Jin et al. [4] developed a vulnerability detection tool to analyze 15,510 PhoneGap apps collected from Google Play. They flagged 478 apps as vulnerable in their experiment.

In Jin et al. paper, they have found a new form of code injection attack which used 14 APIs as code injection channels on HTML5-based Mobile Apps [4]. Then, they developed a vulnerability detection tool to assess the prevalence of the code injection vulnerability in HTML5-based mobile apps. Finally, they have also implemented a prototype called NoInjection as a Patch to PhoneGap in Android to defend against the attack.

Basically, Jin et al. have done a very good job in identifying most of the potential risks imposed by HTML5-based mobile apps in [4]. However, we managed to discover four new vulnerable mobile apps which were not included in their identified vulnerable app list. These four new vulnerable apps could only be detected by our new method because all of them were caused by injecting malicious scripts using HTML5 text box input type along with "document.getElementById ("TagID").value" JavaScript API. However, Jin et al. [4] failed to do so since they only analyzed JavaScript files to identify the injection channels but overlooked HTML files which captured text box input type information.

We implemented our method in a tool called *DroidCIA* will parse HTML files along with JavaScript files to fully analyze the mobile apps. In particular, we will search HTML text box information from those HTML files to create a vulnerable source injection channel list. Then we will parse JavaScript files to see whether they contain any of such injection channels in the above mentioned source list. Subsequently, we will identify the JavaScript control flow and their functional relationship to establish an API call graph such that all the sensitive source-sink pairs could be found by using depth-first search (DFS) algorithm [5].

978-1-4673-7952-6/15 $31.00 © 2015 IEEE
DOI 10.1109/Trustcom.2015.477

We run our detection tool *DroidCIA* on 8303 apps taken from Google Play (more detail discussion in Section 3). In our experiment, we detected 351 vulnerable apps among them there are 4 apps which have text box vulnerable injection channel. The average execution time of our tool on each vulnerable app detection is 38.4 seconds. Our accuracy is 99%.

There are three main contributions made in our new method:

- First, we discovered a new text box injection channel which was not found by other researchers so far because they only analyzed JavaScript APIs but overlooked HTML files which captured text box input type information.
- Second, we provided a tool *DroidCIA* to identify the JavaScript control flow and their functional relationship in order to establish an API call graph such that all the sensitive source-sink pairs could be found by using depth-first search (DFS) algorithm.
- Third, we applied our tool *DroidCIA* to find 347 vulnerable apps like Jin et al. did in [4] and 4 extra vulnerable apps that belonged to this text box injection channel.

The rest of the paper is organized as follows: Section 2 presents a new text box injection channel to inject malicious script which was not detected by other researchers so far. Section 3 describes our detection tool *DroidCIA* which identifies all the code injection vulnerabilities found by both of us and Jin et al. in HTML5-based mobile apps. Section 4 we test *DroidCIA* on 8303 HTML5-based apps obtained from Google Play. Related works are surveyed in Section 5 and the paper is concluded in Section 6.

II. Text Box Injection Channel

Jin et al. [4] proposed many script injection channels and divided them into two categories: the external channel and the internal channel, referring to whether the data come from outside of the device or from inside. However, they didn't consider that an attacker can utilize other external injection channel such as text box injection channel which can input malicious information from the text box input type.

A. The Code Injection Attack

Jin et al. [4] demonstrated a code injection attack on an HTML5-based popular barcode scanner app. They used a malicious string that contains HTML tag and JavaScript code as shown in the following paragraph.

```
1 <img src=x onerror=
2 'navigator.Geolocation.WatchPosition(
3 function(position){
4 info="lat:"+position.coords.latitude+
5 "loc:" + position.coords.longitude;
6 alert(info);
7 b=document.createElement('img');
8 b.src='http:/***.***.**:****?c=' +m })'/>
```

We also used the above malicious string as an example to demonstrate this text box channel code injection attack. The app we used is called `HTTP Position` which was downloaded from Google Play. There is a text box input type available in this app. Therefore, we used it to input the above malicious string. After the string has been accepted, then a window pops up, displaying the users current GPS location as shown in Fig. 1.

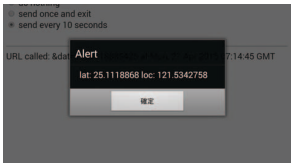

Fig. 1: Get user position attack sample.

B. Using API and HTML TagID to Get TextBox Input String

Getting a text box input string needs to use an API with the TagID for this text box. However TagID and text box information are recorded in an HTML file. In Fig. 2 (b) is a code which shows how TagID ("url") (i.e., Fig. 2 (a)) is written in a HTML file within the `HTTP Position` app. We then use this TagID ("url") along with "document.getElementById ("url").value" External API to receive the malicious string from user input [7].

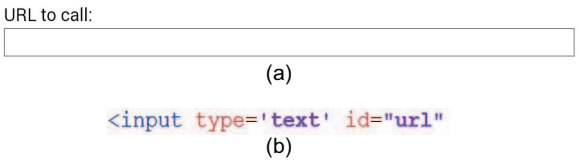

Fig. 2: To get text box position in `HTTP Position` App.

III. Description of DroidCIA

We implemented a Code Injection Attack detection tool named *DroidCIA* to detect code injection attacks on HTML5-based Mobile Apps. Jin et al. have done a very good job in identifying most of the potential risks imposed by HTML5-based mobile apps. However, they didn't consider HTML file which captured text box input type information. We will propose *DroidCIA* which can automate the finding of the new text box injection channel as well as all the risks found by Jin et al. [4].

A. DroidCIA Overview

The scheme of *DroidCIA* is shown in Fig. 3. Our method basically based on Jin's [4] paper. However, we identify new text box Injection Channel and use deep-first search (DFS) algorithm to detect vulnerability apps. We get text box Injection Channel information by using The HTML5 Text Box ID Finder and Injection Channel APIs Creator modules. These modules will search HTML Text Box information from those HTML files to create a vulnerable source injection channel list.

JavaScript of HTML Collector will collect JavaScript information from HTML files. Source API-based Vulnerable

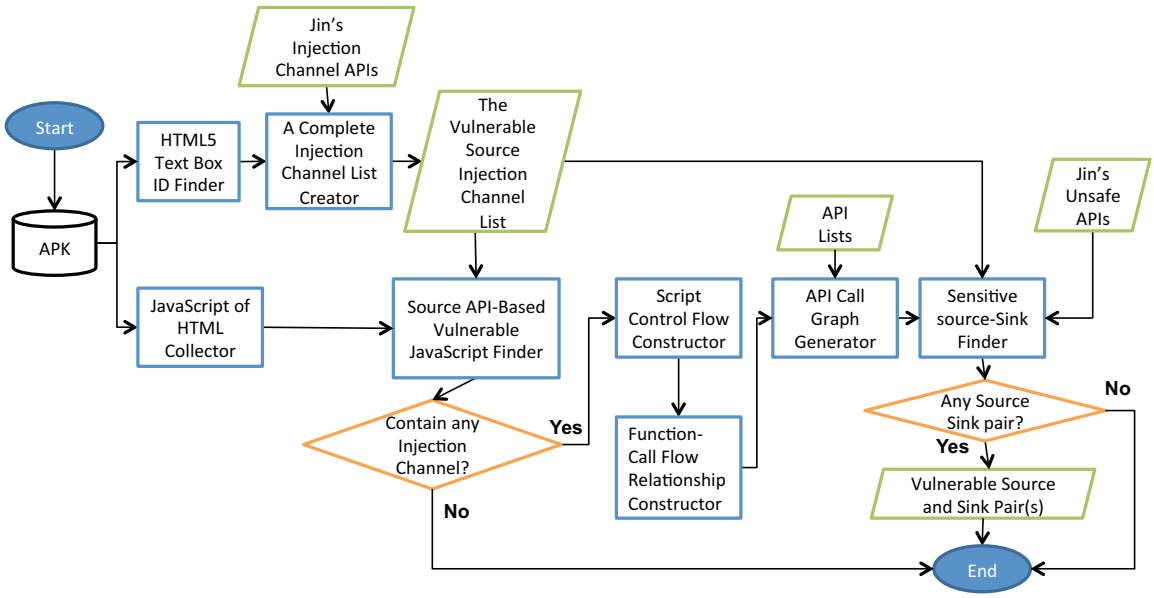

Fig. 3: The *DroidCIA* scheme.

JavaScript Finder will parse JavaScript files to see whether they contain any of such injection channels in The Vulnerable Source Injection Channel List. The last four modules, namely Script Control-Flow Constructor, Function-Call Flow Relationship Constructor, API Call Graph Generator and Sensitive Source-Sink Finder will identify the JavaScript control flow and their functional relationship to establish an API call graph such that all the sensitive source-sink pairs could be found by using depth-first search (DFS) algorithm.

B. HTML5 Text Box ID Finder

In this subsection, we search HTML text box information from those HTML files and use python package BeautifulSoup [4] to find the *id* name(s) from input type tag information with 'text' type. Since each HTML file has its own id list, after each individual file id list has been found, then we associate each file with its own list. Later these combined HTML file and id list information will help us to create The Vulnerable Source Injection Channel List.

C. A Complete Injection Channel List Creator

Jin's et al. [4] 14 Injection Channels didn't have our new text box injection channel so we need to compile a new complete injection channel list. We use the previous id list information to identify those new text box channels which use "document.getElementById('TagID').value" API to inject malicious script. This becomes our new complete The Vulnerable Source Injection Channel List.

D. JavaScript of HTML5 Collector

In this subsection, we collect all JavaScript code information from each HTML file. In HTML5, the tag *script* is used to define a client-side script such as a JavaScript [8]. Moreover, the *script* also contains statements and pointers to an external file through the *src* attribute [8]. Hence, we use python package BeautifulSoup to collect JavaScript information [9]. If *script* is an address of a website then we download the JavaScript information from website. In HTML5, we present three main representations of the *script* such as JavaScript file, JavaScript string in HTML5 and JavaScript from a website.

E. Source API-Based Vulnerable JavaScript Finder

Each HTML5-based application may have many JavaScript files. Our goal is to draw a JavaScript API-call graph to find all the vulnerable source and sink pair(s). However, each JavaScript file may not contain any vulnerable injection channel on the above vulnerable source list. To save the processing time we only process further those JavaScript files, which contain at least one injection channel on such list and skip those which contain no such injection channel.

F. Script Control-Flow Constructor

To identify the JavaScript control flow, we use a tool named as Type Analyzer for JavaScript (TAJS) [10]. Each node means an instruction and each edge presents control-flow between the instructions in the program [10]. For example, the code *alert("onload")* in function *onLoaded* shown in Fig. 4. We use TAJS to obtain the control-flow graph and several information such as Function Name, Execution Sequence (e.g., Edge), Command Parameter (e.g., Block) and Parameters Attribute as shown in Fig. 5

All above information are stored in a *.dot* file. From this *.dot* file, we can obtain Function Name, Execution Sequence, Command Parameter and Parameters Attribute. Furthermore for those information missing in *.dot* file we will resolve this problem to extract the necessary line number information.

```
function onLoaded()
{
alert("onloaded");
document.addEventListener("deviceready",
onDeviceReady, false);
}
```

Fig. 4: JavaScript Sample Code.

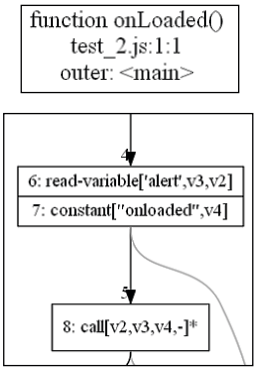

Fig. 5: The Script Control-Flow of JavaScript Sample Code.

G. Function-Call Flow Relationship Constructor

To identify JavaScript function-call flow relationship, we need to extract all the relevant function-call information from Script Control-Flow. Additionally, we also need to perform two consecutive tasks, namely, "Declarative Function Name Finding" and "Function-Call Relationship Finding".

Declarative Function Name Finding The declarative function names in JavaScript are usually shown in the $< main >$ function of the Script Control-Flow. For example, in Fig. 6 (b), there are two declarative function names, namely, *onLoaded* and *onDeviceReady* in JavaScript code. For example, in Fig. 6 (b), there are two declarative function names, namely, *onLoaded* and *onDeviceReady* in JavaScript code. Referring to Fig. 6 (a) we can see two functions *onLoaded* and *onDeviceReady* are declared in the $< main >$ function of the Script Control-Flow file. Subsequently, we construct two starting function-call relationship nodes *onLoaded* and *onDeviceReady*.

Function-Call Relationship Finding Previously, we have already identified *onLoaded* and *onDeviceReady* as two starting nodes in the function-call relationship charts. There are three sub-tasks involved to complete the function-call relationship construction, namely, Expressed Sub-Function-Call Finding, PhoneGap API-Function-Call Finding and JavaScript Function-Call Finding.

1) **Expressed Sub-Function-Call Finding**
The Script Control-Flow has a type called *function-expr* in each node which identifies it as a function type (e.g., Fig. 7 (a)). For example, in Fig. 7 (b), onDeviceReady function will call *function(result)* and *function(error)*. Hence, we can cross reference with Script Control-Flow

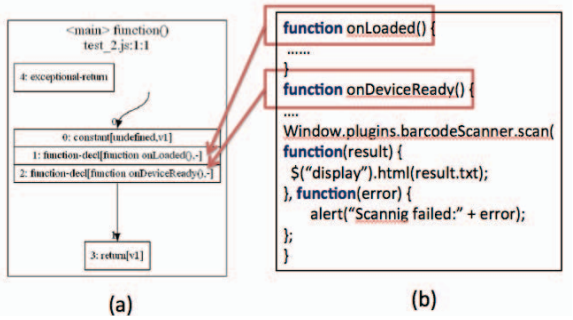

Fig. 6: How to do declarative function name finding on Script Control Flow. (a) Script Control-Flow $< main >$ function. (b) JavaScript Sample Code.

to construct a Sub-Function-Call relationship in Fig. 7 (c).

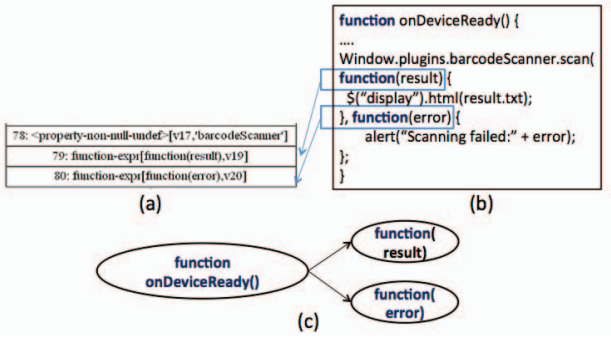

Fig. 7: How to perform expressed sub-function call finding. (a) It mentions the Script Control-Flow of function *onDeviceReady*. (b) The example of function code in *onDeviceReady*. (c) The drawing of relation between two sub-functions and *onDeviceReady()*.

2) **PhoneGap API-Function-Call Finding**
Some PhoneGap APIs have function call which doesn't exist in Script Control-Flow since PhoneGap APIs have their unique function-call characteristics. We modelled PhoneGap API function call like Jin et al. [4] to construct the function call graph. For example, as shown in Fig. 8 (a) the function *onLoaded* has a PhoneGap API "document.addEventListener("deviceready",onDeviceReady, false)", which invokes *onDeviceReady* function. Therefore, we need to create a function call relationship between *onLoaded* and *onDeviceReady* (e.g., Fig. 8 (b)).
3) **JavaScript Function-Call Finding**
Please note Script Control-Flow doesn't have complete function call information since some JavaScript function calls are extracted from that were labelled as variable type instead of function-call type. For example, there is a JavaScript code in Fig. 9 (b). Furthermore, in Fig. 9 (a), *setAudioPosistion* is a *read-variable* type

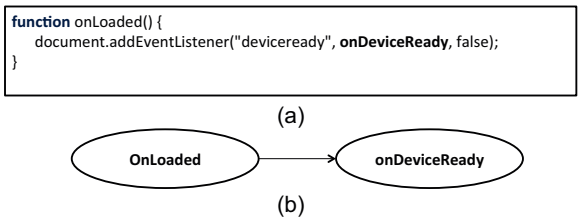

Fig. 8: How to perform PhoneGap API-function call. (a) The example of function code in *onLoaded()*. (b) The example of relation between *onLoaded* and *onDeviceReady*.

instead of *function-expr* type. Therefore, we must use JavaScript Graph tool [11] [12] to create this function relationship. JavaScript Graph tool is a website page, and we downloaded this website page and modified JavaScript in html file to get function call information. Fig. 10 showed that we able to get the function call result using JavaScript Graph tool.

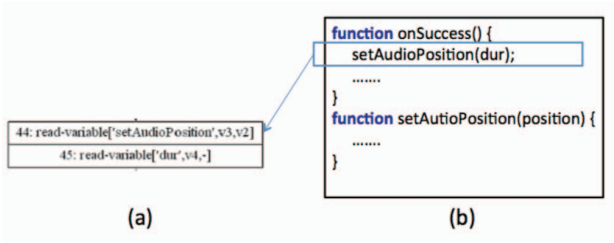

Fig. 9: Incomplete function call information sample. (a) In control-flow doesn't label setAudioPosistion for function (b) The example of incomplete function code.

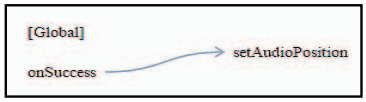

Fig. 10: The drawing of JavaScript Function Call

H. API Call Graph Generator

To establish the API call graph, we use Script Control-Flow, Function-Call Relationship and a comprehensive API List. This comprehensive API List consists of: HTML DOM APIs (e.g., document.addEventListener) [13], jQuery APIs (e.g., height, hide, html and append) [14] and PhoneGap APIs (e.g., window.plugins.barcodeScanner.scan) [15]. In general, we extract a function API call graph for each node which is contained in the Script Control-Flow file by cross-referencing with a comprehensive API List. Then, we connect those function API call graphs which have function-call relationship with one another using the results generated from the Function-Call Relationship Constructor.

The details of API Call Graph Generator are described as follows:

1) **Create a function API call graph**
In Script Control-Flow, each node contains an instruction and each edge represents potential control-flow between instructions in the program [10]. Hence, we discover all nodes which are APIs and their relation. We show an example code in Fig. 11 (a) of construction of API subgraph from each function (e.g., Fig. 11 (b)).

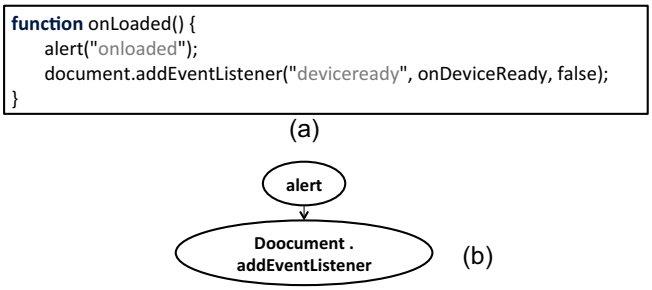

Fig. 11: Create API subgraph from each function sample. (a) The example code of create API subgraph. (b) The drawing of create API subgraph.

2) **Connect API subgraphs through function-call relation.**
Now, there are all API call subgraphs now, and then we need to connect these API call subgraphs based on function relation information. After connecting API call graphs, we also need to remove the redundant relation between original functions. We present the process of edge add and edge remove in Fig 12 and Fig 13. For example, we have two function relations described as follow:
- function onLoaded() → function onDeviceReady()
- function onDeviceReady() → function(result)

After adding function call, we need to remove redundant edge between $barcodeScanner.scan$ and $alert$. Finally, we get the result like Fig. 13. We can acquire the complete API-call Graph of a HTML5-based mobile application. Furthermore, we can use it to find vulnerable sequence(s).

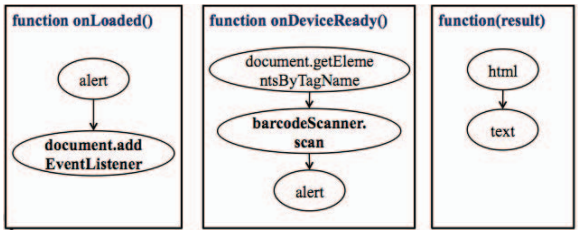

Fig. 12: Creating API subgraph from each function

I. Sensitive Source-Sink Finder

In this subsection, our goal is to find the vulnerable Source API and vulnerable Sink API from API Call Graph. We use depth-first search (DFS) algorithm to find all possible

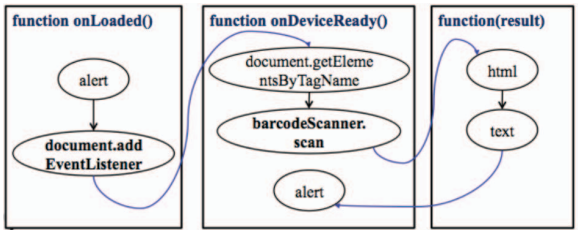

Fig. 13: Connection API subgraphs through functionl relation.

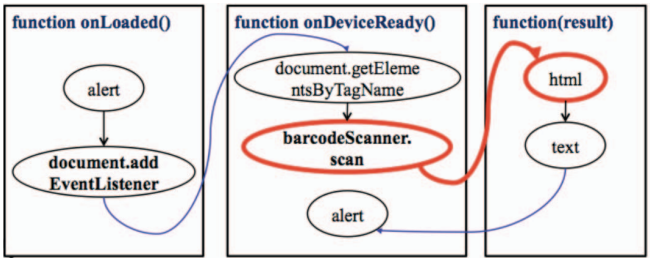

Fig. 14: The example of sensitive source-sink finding.

vulnerable paths. We can ensure if path has one pair of Source API and Sink API then this app is vulnerability.

The Source Injection Channel API, we use Jin et al. [4] Injection Channel APIs and our text box injection channel ("TagID" is the HTML Tag ID name). Unsafe API (DOM - Document Object Model), we use Jin et al. [4] Unsafe API. We can see those APIs in Table I.

TABLE I: The vulnerable source and sink APIs

Source Injection Channel API (Source API)	External API	NFC.addNdeListener
		NFC.addMimeTypeListener
		NFC.addTagDiscoveredListener
		Bluetooth.getUuids
		WifiInfo.get
		barcodeScanner.scan
		FileTransfer.download
		document.getElementById ('TagID').value
	Internal API	Contacts.find
		DirectoryEntry.getDirectory /getFile
		DirectoryReader.readEntries
		Entry.getMetadata
		FileEntry.file
		FileReader.readAnText
		Media.getFormatData
Unsafe API (Sink API)	DOM API	document.write()
		document.writeln()
		innerHTML
		outerHTML
	jQuery API	html()
		append()
		prepand()
		before()
		after()
		replaceAll()
		replaceWith()

We use python DFS package to implement parse of API Call Graph [16]. If the vulnerable path is found, we will output the apk name and pair of source API and sink API. We present a sample of vulnerable sequence in Fig. 14. We find the vulnerable path with source API $barcodeScanner.scan$ and vulnerable sink API $html$.

IV. Evaluation

We conducted an evaluation on a data set similar to Jin et al. used in [4]. We applied this new *DroidCIA* method to a vulnerable app set with 8303 apps obtained from Google Play like Jin et al. did. In order to optimize *DroidCIA's* performance, we limit the number of nodes in Script Control-flow Graph to less than 20000 and the average run-time to 38.4 seconds for each application.

A. Dataset

We were only able to download 8303 HTML5-based apps from the same Google Play web site provided to us by Jin et al. However, in their paper [4], there were 478 vulnerable apps found in their original test data set of 15,510 apps. While among the previously downloaded 8303 apps we were only able to download 347 vulnerable apps among the original 478 apps found by Jin et al. because some of them were already removed from Google Play or were fixed by the developers.

B. Accuracy

In this evaluation, our *DroidCIA* method detected a total of 351 vulnerable apps which contains 4 more apps than 347 that were detected by Jin et al. Please note that *DroidCIA* was able to detect all the 347 vulnerable apps that can be detected by Jin et al. [4]. In addition, we were also able to detect 4 extra vulnerable apps that belonged to the text box injection channel. The accuracy rate of our method is 99.21%. We discovered 64 apps were false-positive applications. We manually checked these false-positive samples and found that the input strings of text box were benign. However, the sensitive source and sink still existed in the control-flow graph, which effected our detection. Additionally, dead code never happened when we performed the vulnerable apps detection, which also effected our detection. These problems need to be fixed by dynamic analysis in the future.

C. Case Studies

We proposed a number of vulnerable source and sink APIs for these 351 vulnerable apps. Fig. 15 (a) shows the Injection Channel API usage, while Fig. 15 (b) shows the Unsafe API usage. There are 4 apps that have text box injection channel. These source and sink APIs information are captured in Table II, while text box injection channel attack sample was covered in Section II.

TABLE II: Vulnerable Apps Detail Information

Apk Name	Source API	Sink API
br.com.cherobin .acate.apk	document.getelem entbyid('searchbox') .value	innerHTML
it.rmdtmsoft .nccfirenze.apk	document.getelem entbyid('inputloca') .value	append
leadtools.medical webviewerdemo.apk	document.getelem entbyid('aetitle') .value	html
org.apache.cordova .http_position.apk	document.getelem entbyid('url') .value	innerHTML

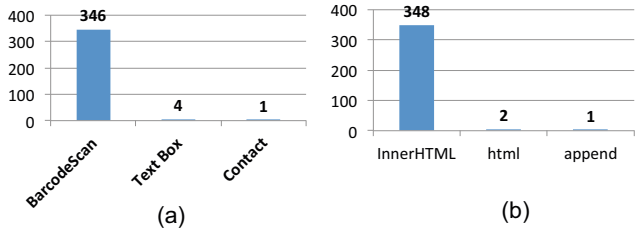

Fig. 15: The Vulnerable API usgae. (a) Injection Channel APIs usage. (b) Unsafe APIs usage.

D. Limitations

Our evaluation shows that *DroidCIA* is effective in detecting the vulnerable apps. However, we admit that there are still several limitations in our method which requires further improvement. After we manually checked the vulnerable applications, we discovered two key problems and other miscellaneous problems as follows:

Input String Problem
The vulnerable sequence can't ensure whether the input string from source API is malicious or not. We found some source APIs could contain the locked string that is not inputted by the user.

Dead Code Problem
We found some vulnerable sequences hidden as dead code but they will never be triggered.

Other Miscellaneous Problems
Our function call graph may still miss some asynchronous function calls. Some vulnerable sequence(s) may not be found by our method.

In our future work we decide to use dynamic analysis to improve current static analysis shortcomings.

V. Related Work

HTML5-based apps security can't be ignored. To better understand the HTML5-based apps security research, we surveyed both HTML5-based and WebView related papers, because HTML5-based apps make use of WebView to embed a web browser component.

Jin et al. [4] studied the potential risks imposed by HTML5-based mobile apps. They implemented a tool to detect vulnerable apps on a data set downloaded from Google Play. But they didn't consider the HTML user interaction tag information. Our method considered the HTML user interaction and found 4 extra vulnerable apps that belonged to the new Text Box injection channel. Martin et al. [17] discovered that the Web security model and the local security model are not coherent. Furthermore, they also found even minor bugs in either the hybrid code, or the embedded browser could open the door to cross-site scripting attacks. They presented NoFrak, a capability-based defense against fracking attacks. However, Martin et al. only focused on fixing fracking attack and failed to consider the code injection from user interaction, barcode, TextBox...etc. Jin et al. [18] studied the potential risks of HTML5-based applications, and investigated how the existing mobile systems' access control could support these applications.

WebView is an important component of HTML5-based application. Luo et al. [19] discussed a number of attacks on WebView. They have identified two fundamental causes of these attacks: weakening of the TCB and sandbox. But they didn't provide any defense against the attacks on WebView. Shin et al. [20] presented an approach for supporting visual security cues to prevent the SSLstripping attack in WebView-based apps. Their implementation could be improved by introducing additional features such as dynamic evaluation of the submission form so that it can detect any malicious activity after the page is rendered. Wang et al. [21] reported the first systematic study on this mobile cross-origin risk. Luo et al. [22] pointed out that even if those APIs were secured, WebView would still be vulnerable. Chin et al. [23] explored two WebView vulnerabilities: excess authorization, where malicious JavaScript could invoke Android Application code; and file-based cross-zone scripting, which could expose a device's file system to an attacker. Yu. et al. [24] studied a case of an attack from web pages.

In order to analyze JavaScript files we surveyed several JavaScript analysis papers. Jensen et al. [4] presented a static program analysis infrastructure that could infer detailed and sound type information for JavaScript programs using abstract interpretations. This paper helps us to analyze JavaScript programs. Jensen et al. [25] discussed the security challenges, which included the dynamic aspects of JavaScript and the complex interactions among JavaScript, HTML and the browser. They presented the first static analysis that was capable of reasoning about the flow of control and data in modern JavaScript applications that interacted with the HTML DOM and browser API. Feldthaus et al. [26] presented a scalable field-based flow analysis for constructing call graphs.

VI. Conclusion

In this paper, we propose a new injection channel for HTML5-based application that could be used by an attacker to inject malicious script code. It is called a new text box injection channel. This can help us detect more vulnerable

apps. We detected a total of 351 vulnerable apps with 99% accuracy which included 347 detected also by other researchers as well as 4 extra vulnerable apps that belonged to this text box injection channel. Although text box injection channel occurred ratio doesn't often, text box using is very popular, so this injection channel detection we can't be ignored. The attacker also can write the malicious script in the app and let the user trigger the malicious script in the user does not know the situation. In our future work, we plan to use dynamic API trace to find the vulnerable API sequence in order to detect not only PhoneGap but also other HTML5-based apps.

ACKNOWLEDGMENT

This research is supported in part by the National Science Council of Taiwan under grants number NSC 102-2218-E-011-010 MY3 and 102-2218-E-011-011 MY3. The authors also gratefully acknowledge the helpful comments and suggestions of the reviewers, which have improved the presentation.

REFERENCES

[1] *Worldwide Smartphone Shipments Increase 25.2Heightened Competition and Growth Beyond Samsung and Apple, Says IDC*. [Online]. Available: http://www.idc.com/getdoc.jsp?containerId=prUS25224914

[2] *75 of Developers Using HTML5: Survey*. [Online]. Available: http://www.eweek.com/c/a/Application-Development/75-of-Developers-Using-HTML5-Survey-508096/

[3] *China Mobile Internet: Fragmentation and Shorter Life Cycle*. [Online]. Available: http://tip.umeng.com/uploads/data_report/2013firsthalf_eng.pdf

[4] 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 *ACM Conference on Computer and Communications Security (CSS)*, 2014.

[5] J. H. Reif, "Depth-first search is inherently sequential," in *Information Processing Letters*, 1985.

[6] *HTML id attribute*. [Online]. Available: http://www.w3schools.com/tags/att_global_id.asp

[7] *The HTML DOM Attribute Object*. [Online]. Available: http://www.w3schools.com/jsref/dom_obj_attributes.asp

[8] *HTML Script Tag*. [Online]. Available: http://www.w3schools.com/tags/tag_script.asp

[9] *BeautifulSoup Package*. [Online]. Available: https://pypi.python.org/pypi/beautifulsoup4/4.3.2

[10] S. H. Jensen, A. Moller1, and P. Thiemann, "Type analysis for javascript." in *In Static Analysis Symposium (SAS)*, 2009.

[11] *UglifyJS*. [Online]. Available: https://github.com/mishoo/UglifyJS

[12] *JavaScrip Function Graph*. [Online]. Available: http://jfg.atwebpages.com/

[13] *JavaScript and HTML DOM Reference*. [Online]. Available: http://www.w3schools.com/jsref/

[14] *jQuery API*. [Online]. Available: http://api.jquery.com/

[15] *PhoneGap API*. [Online]. Available: http://docs.phonegap.com/en/4.0.0/index.html

[16] *Depth-First Search and Breadth-First Search in Python*. [Online]. Available: http://eddmann.com/posts/depth-first-search-and-breadth-first-search-in-python/

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

[18] X. Jin, L. W. andTongbo Luo, and W. Du, "Fine-grained access control for html5-based mobile applications in android." in *In Proceedings of the 16th Information Security Conference (ISC)*, 2013.

[19] 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*, 2011.

[20] D. Shin, H. Yao, and U. Rosi, "Supporting visual security cues for webview-based android apps," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, 2013.

[21] R. Wang, L. Xing, X. Wang, and S. Chen, "Unauthorized origin crossing on mobile platforms: threats and mitigation," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. Pages 635–646.

[22] T. Luo, X. Jin, A. Ananthanarayanan, and W. Du, "Touchjacking attacks on web in android, ios, and windows phone," in *Foundations and Practice of Security. Springer Berlin Heidelberg*, 2013.

[23] E. Chin and D. Wagner, "Bifocals: Analyzing webview vulnerabilities in android applications," in *Proceedings of the 14th International Workshop on Information Security Applications*, 2013.

[24] J. Yu and T. Yamauchi, "Access control to prevent attacks exploiting vulnerabilities of webview in android os," in *Processing of the 2013 IEEE International Conference on High Performance Computing and Communications (HPCC13)*, 2013.

[25] S. H. Jensen, M. Madsen, and A. Mller, "Modeling the html dom and browser api in static analysis of javascript web applications," in *Proceedings of the 16th Information Security Conference (ISC)*, September 2011.

[26] A. Feldthaus, M. Schfer, M. Sridharan, J. Dolby, and F. Tip, "Efficient construction of approximate call graphs for javascript ide services," in *Proceedings of the 2013 International Conference on Software Engineering*, 2013, pp. Pages 752–761.