Detection and Prevention of Code Injection Attacks on HTML5-based Apps

Xi Xiao[a], Ruibo Yan[a], Runguo Ye[b], Qing Li[a], Sancheng Peng[c], Yong Jiang[a]
[a] Graduate School at Shenzhen, Tsinghua University, Shenzhen, China
[b] China Electronics Standardization Institute, Beijing, China
[c] School of Computer Science, Zhaoqing University, Zhaoqing, China
Email: xiaox@ sz.tsinghua.edu.cn, yrb15@mails.tsinghua.edu.cn, wudi@isccc.gov.cn, li.qing@sz.tsinghua.edu.cn, psc346@aliyun.com, jiangy@ sz.tsinghua.edu.cn

Abstract—**Security on mobile devices is becoming increasingly important. HTML5 are widely used to develop mobile applications due to its portability on multi platforms. However it is allowed to mix data and code together in web technology. HTML5-based applications are prone to suffer from code injection attacks that are similar to XSS. In this paper, at first, we introduce a more hidden type of code injection attacks, coding-based attacks. In the new type of code injection attacks, JavaScript code is encoded in a human-unreadable form. Then we use classification algorithms of machine learning to determine whether an app suffers from the code injection attack or not. The experimental result shows that the *Precision* of our detection method reaches 95.3%. Compare with the other method, our approach improves a lot in detection speed with the precision nearly unchanged. Furthermore, an improved access control model is proposed to mitigate the attack damage. In addition, filters are adopted to remove JavaScript code from data to prevent the attacks. The effectiveness and rationality have been validated through extensive simulations.**

Keywords—code injection; classification algorithm; machine learning; access control model; filter

I. Introduction

Mobile devices are becoming increasingly popular. Many attackers have focused on mobile applications(apps). Security on mobile devices is more important[1,2]. HTML5[3] technology is used widely because of its portability on multi platforms. An increasing number of applications are developed by HTML5, CSS and JavaScript. Especially when middlewares, such as PhoneGap, come out, many developers adopt HTML5 technology to develop mobile applications that have the same functions as native apps. But in web technology, it is allowed to mix data and code together. Thus HTML5-based apps can easily suffer from code injection attacks[4,5] that are similar to Cross Site Scripting (XSS). The IndexedDB is introduced in HTML5, which brings a new distributed approach to processing big data. So solving security problems of HTML5-based apps can benefit security on big data.

In all the middlewares, PhoneGap is a typical one and is widely used. It is a free and open source framework that allows us to create mobile apps for the platforms you care about with standardized web APIs[6]. Developers can write code that runs on multi platforms using web technologies through PhoneGap. Since Android OS is open source and widely used in thousands of and millions of smartphones, in this paper, we focus on analyzing Android, but the idea can also be extended to other platforms. The source code of PhoneGap contains two parts: JavaScript framework and Java Native framework. Java Native framework consists of the bridge part and the plugins part and it is developed by native language to access the resources of mobile devices, such as contacts, SMS, camera etc.

The code injection attack is similar to XSS. XSS is a type of computer security vulnerability typically in web applications. XSS enables attackers to inject client-side JavaScript code into web pages which are viewed by other users[7]. When other users view these web pages, the injected JavaScript code will be executed. Through XSS, the attackers can collect information of the client environment, steal user's cookies, redirect to other web pages and so on. XSS mostly happens in web applications, but code injection attacks happen in mobile applications. Attackers can inject malicious code into contacts, SMS, barcode and meta information of files. Once these data are displayed by applications written by PhoneGap, the injected code will be triggered. In this way, code injection attacks happen.

The code injection attack is firstly proposed in [4], but the JavaScript code is injected in plain text. In this case, the injected code is not hidden and application users can recognize injected code easily. [5] presented a detection method using static analysis to find call function sequences. But the time complexity is very high. It takes 15.38 seconds averagely to detect whether an app is vulnerable for the attack.

To solve the problems mentioned above, at first, we introduce coding-based code injection attacks which encode the JavaScript code in a human-unreadable format. Furthermore, we extract permissions, JavaScript functions in PhoneGap JavaScript frameworks and unsafe JavaScript APIs as the features. Then nine machine learning methods are used to classify whether an application is vulnerable. 578 normal apps and 408 vulnerable apps are downloaded from Google Play as the experimental data. In the experimental results, our method costs no more than 2 seconds to detect one application and the precision can achieve up to 95.3%. In addition, we improve an existing access control model to mitigate the

This work is supported by the NSFC projects (61202358, 61371078, 61379041), the National Basic research Program of China (2012CB315803), the National High-Tech R&D Program of China (2014ZX03002004), the Research Fund for the Doctoral Program of Higher Education of China (20130002110051) and the Shenzhen Key Laboratory of Software Defined Networking.

damage. Filters are also adopted to remove the JavaScript code from data to prevent applications from attacks. There are mainly three contributions in our work. Firstly, we propose a coding-based code injection attacks. It is more hidden than existing code injection attacks. Secondly, we put forward a new detection method to recognize whether an application is vulnerable for suffering from code injection attacks. Compared with the other method of [5], our approach improves a lot in detection speed with the precision nearly unchanged. Thirdly, we improve an existing access control model by adding permissions in HTML file and adding filters in Android source code. The effectiveness and rationality of the proposed method have been validated through extensive simulations.

Machine learning is used widely in big data, we also use classifiers of machine learning in this paper. The remainder of this paper is structured as follows: In Section 2, we review the existing code injection attacks and propose a new type of code injection attacks. Detection of code injection attacks is discussed in Section 3. In Section 4, we present methods to prevent code injection attacks. We provide an overview of related work in Section 5 and conclude the paper in Section 6.

II. Code Injection Attacks

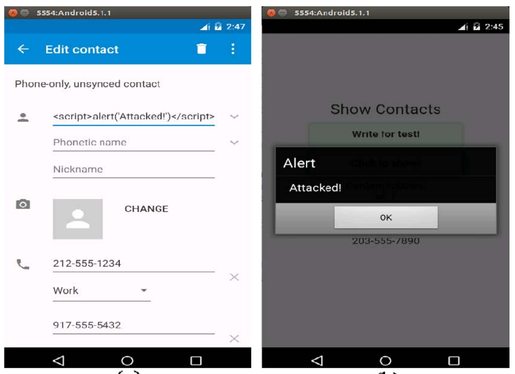
A. Code Injection Attacks already Proposed

The code injection attack is raised firstly in [4]. It is allowed to mix code and data together in web technologies. Applications that use PhoneGap framework or other middlewares will suffer from code injection attacks. Attackers add malicious code to the data, and submit the data to the mobile application with textbox. Since the application is developed by HTML5 technologies, it is truly a web page. The injected code will be parsed and executed by the JavaScript engine of WebView[8]. Consequently, code injection attacks will happen.

Let's take an example to show how code injection attack happens. We develop an app using PhoneGap framework named ShowContacts. The application can write three test data to the system contacts and show all the records of system contacts.

As Fig. 1 shows, the main interface of the application has two buttons, one button used to write test data and the other used to show all contact records. We add an contact whose display name is "<script>alert('Attacked!')</script>". Then we use our application to show all the contacts. The injected code, i.e. the display name, will be triggered. The application will pop up an window displaying "Attacked!". This is how code injection works.

There are many code injection channels[5]. Mobile devices can get data from many external channels such as scanning barcodes by camera, receiving SMS messages, reading NFC. Attackers can also inject malicious code into the metadata of multi-media files, such as the title, the artist album of a song or video. WIFI SSID and Bluetooth id can also be used to inject malicious code.



(a) (b)

Fig. 1. Code Injection Attack Example

B. New Coding-based Code Injection Attacks

There are many types of encoding in XSS. Similarly, there are many types of encoding in code injection attacks. There are two types of encoding for injected JavaScript code, HTML coding and JavaScript coding.

HTML Coding. HTML coding includes HTML entity coding and HTML n-coding. For example, the JavaScript code is shown below:

document.write("<script>alert('Attacked')</script>");

Its size is 53. After HTML entity coding, it turns into

document.write(&quot;&lt;script&gt;alert(&#39;Attacked&#39;)&lt;/script&gt;&quot;);

The size of code after coding is 83.

HTML n-coding usually contains HTML decimal coding and HTML hexadecimal coding. HTML entity coding is a coding format that contains characters like "&lt;&quot&gt;". With respect to HTML decimal coding, if you get the ASCII number of a character, and add &# in front of the ASCII number, you get the decimal coding. For example, the code is shown below:

<img src="x" onerror="alert('A')">

Its size is 34. After HTML decimal coding, it turns into

<imgsrc="x"onerror="&#97;&#108;&#101;&#114;&#116;&#40;&#39;&#65;&#39;&#41;">

Its size is 78.

If using HTML hexadecimal coding, the code turns into

<imgsrc="x"onerror="&#x61;&#x6C;&#x65;&#x72;&#x74;&#x28;&#x27;&#x41;&#x27;&#x29;">

Its size is 84.

We do many experiments and find that all the code processed by HTML entity coding or HTML n-coding can cause code injection attacks.

JavaScript Coding. JavaScript coding has mainly five types: Unicode coding, hexadecimal coding, octal coding, escape coding and Base64 coding. For example, the JavaScript code is shown below:

<script>alert('Attacked')</script>

Its size is 34. After Unicode coding, the code turns into

\u003c\u0073\u0063\u0072\u0069\u0070\u0074\u003e\u0061\u006c\u0065\u0072\u0074\u0028\u0027\u0041\u0074\u0074\u0061\u0063\u006b\u0065\u0064\u0027\u0029\u003c\u002f\u0073\u0063\u0072\u0069\u0070\u0074\u003e

Its size is 34. After hexadecimal coding, the code turns into

u003c\u0073\u0063\u0072\u0069\u0070\u0074\u003e\u0061\u006c\u0065\u0072\u0074\u0028\u0027\u0041\u0074\u0074\u0061\u0063\u006b\u0065\u0064\u0027\u0029\u003c\u002f\u0073\u0063\u0072\u0069\u0070\u0074\u003e

Its size is 34. After octal coding, the code turns into

\74\163\143\162\151\160\164\76\141\154\145\162\164\50\47\101\164\164\141\143\153\145\144\47\51\74\57\163\143\162\151\160\164\76

Its size is 34. After escape coding, it turns into

\<script\>alert(\'Attacked\')\<\/script\>

Its size is 34. After base64 coding, the code turns into

PHNjcmlwdD5hbGVydCgnQXR0YWNrZWQnKTwvc2NyaXB0Pg==

Its size is 48.

Our experimental results show that the JavaScript code after encoding can be executed except base64 coding.

Coding Types Suitable for Code Injection. Next we will investigate whether an coding format is suitable for injecting code. First, for the injected code, it is used to complete specific functions. The smaller its size is, the better the case is. For the three types of HTML coding, the size of injected codes becomes bigger. So it is not good for injecting codes. Second, the JavaScript code after coding needs an function to trigger. That is to say, you cannot encode the injected code completely. For example, with respect to such slice of JavaScript code

document.write("<script>alert('Attacked')</script>"),

we can only encode "<script>alert('Attacked')</script>" because "document.write()" is a trigger condition. If the trigger condition function is also encoded, code injection attacks will not happen. So the three types of HTML coding is not suitable for code injection.

As for the five types of JavaScript coding, the JavaScript code after base64 coding cannot be executed. It is not suitable for code injection. Although the size of code remains the same after escape coding, the JavaScript code is not hidden. Some users or developers can easily understand what the code means. Thus escape coding is not suitable for code injection. Since the size of code remains the same after the other three types of encoding, as well as the code after encoding is more hidden, JavaScript Unicode coding, JavaScript hexadecimal coding and JavaScript octal coding are suitable for code injection.

How Code Injection Attack Happens. In order to trigger injected code, the injected code and data must be displayed in the HTML page. There are two conditions to make code injection attack happen. Firstly, the application need to read data from external channels. Secondly, the data read from external channels must be displayed in HTML pages. Also the way of displaying data must use unsafe JavaScript APIs[5]. The application read data from external channels through the plugins of PhoneGap framework. Table I shows the popular plugins of PhoneGap, its functions that read data from external channels and its permissions needed on Android OS.

TABLE I. POPULAR PHONEGAP PLUGINS

Plugin packagename	Plugin functions	Permissions of plugin

com.phonegap.plugins.barcodescanner	cordova.plugins.barcodeScanner.scan()
	cordova.plugins.barcodeScanner.encode()
		CAMERA
		FLASHLIGHT

org.apache.cordova.contacts	navigator.contacts.create()
	navigator.contacts.find()
	navigator.contacts.pickContact()
		READ_CONTACTS
		WRITE_CONTACTS
		GET_ACCOUNTS

com.cordova.plugins.sms	sms.send()	SEND_SMS READ_PHONE_STATE

com.chariotsolutions.nfc.plugin	nfc.addNdefListener()
	nfc.removeNdefListener()
	nfc.addTagDiscoveredListener()
	nfc.removeTagDiscoveredListener()
		NFC

org.apache.cordova.file	cordova.file.applicationDirectory()
	cordova.file.applicationStorageDirectory()
	cordova.file.documentsDirectory()
		WRITE_EXTERNA_STORAGE

com.connectivity.monitor	connectivity.observeRemoteIPV4()
	connectivity.observeRemoteIPV6()
	connectivity.observeLocalWifi()
		ACCESS_NETWORK_STATE
		INTERNET

com.megster.cordova.bluetoothserial	bluetoothSerial.connect()
	bluetoothSerial.connectInsecure()
	bluetoothSerial.disconnect()
		BLUETOOTH
		BLUETOOTH_ADMIN

org.apache.cordova.battery-status	batterystatus batterycritical batterylow	None

III. DETECTION OF VULNERABLE APPS

The way to detect a malicious application is usually divided into two types: static analysis and dynamic analysis. Static analysis is performed without executing programs. When doing static analysis, in order to find abnormal behaviors, analysts usually write an automated system to read and scan the source code of an application. While dynamic analysis is performed by executing programs on a real environment. The target application is executed with sufficient test inputs to produce interesting behavior.

In Section 2, How Code Injection Attack Happens, it is introduced that applications suffering from code injection attacks usually have sources and sinks. As for the application developed by PhoneGap, sources are the functions of PhoneGap plugins that read data from external channels and sinks are functions that display data in HTML page. In our case, sinks are the unsafe APIs of JavaScript and jQuery. So we choose sources and sinks as part of features.

In this paper, we use static analysis to detect whether an application is vulnerable. The main idea is to extract permissions, PhoneGap plugin functions and JavaScript unsafe APIs as features, and then use these features to train the classifier to make predictions. We use a tool named Weka to classify. Weka is a collection of machine learning algorithms for data mining tasks[9].

The detection procedure is shown in Fig. 2

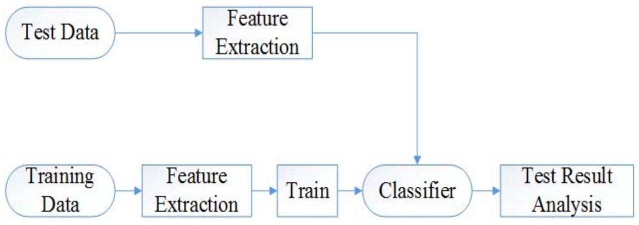


Fig. 2. Detection Procedure

In our detection method, 300 normal apps and 300 vulnerable apps are used to train to build a classifier. 278 normal apps and 108 vulnerable apps are used to test the classifier. Then comes the test result analysis.

The detail steps of our detection method are as follows.

A. *Preprocessing*

As we mentioned before, we use the permissions of an application, JavaScript functions of PhoneGap plugins to read data from external channels and unsafe JavaScript APIs as features of classification algorithms. The permission features extracted are shown in Table II. When developers use a specific PhoneGap plugin, they need to apply specific permissions in Android Manifest file. Table II shows the permissions and corresponding PhoneGap plugin channels.

The reason to choose these permissions is that these permissions are applied by PhoneGap plugins. Also these permissions are usually own by malicious applications[10]. PhoneGap JavaScript functions that read data from external channel are shown in Table I. Table III shows the string we use to judge whether a PhoneGap JavaScript function is used in an application. For example, if string "sms.send(" is found in HTML or JavaScript files of an app(excluding the framework files, such as jQuery, Cordova.js and so on), we assume that this app used "sms.send()" function of PhoneGap library. In the preprocess procedure, we use the string of "String of PhoneGap JavaScript functions" of Table III to recognize whether a function is used in an application. Unsafe JavaScript APIs to display data can be found in [5].

TABLE II. PERMISSION FEATURES

Permission	Channel
android.permission.INTERNET	Default
android.permission.ACCESS_NETWORK_STATE	Wi-Fi
android.permission.CAMERA	Barcode
android.permission.FLASHLIGHT	Barcode
android.permission.ACCESS_COARSE_LOCATION	Geology
android.permission.ACCESS_FINE_LOCATION	Geology
android.permission.WRITE_EXTERNAL_STORAGE	File
android.permission.RECORD_AUDIO	Muti-media
android.permission.RECORD_VIDEO	Muti-media
android.permission.MODIFY_AUDIO_SETTINGS	Muti-media
android.permission.READ_PHONE_STATE	Phone state
android.permission.READ_CONTACTS	Contacts
android.permission.WRITE_CONTACTS	Contacts
android.permission.GET_ACCOUNTS	Contacts

We use command "aapt" to get the permissions of an application. And we employ string matching to determine whether a permission or a function is used in an application. If an application has the specific feature, we use number 1 to represent it. Otherwise, we use 0 to represent it.

TABLE III. PHONEGAP JAVASCRIPT FUNCTIONS

String of PhoneGap JavaScript functions	Channel
cordova.plugins.barcodeScanner.	Barcode
sms.send(	SMS
nfc.	NFC
cordova.file.	File
connectivity.	Wi-Fi
bluetoothSerial.	Bluetooth
navigator.contacts.	Contacts
battery	Battery

In this paper, we use Ubuntu 14.04.2, 8G RAM, Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz. The time of preprocessing 578 normal apps is 159.168612957 seconds. It takes 80.5510399342 seconds to preprocess 408 vulnerable apps. The format of the preprocessing result is shown below.

1,1,1,0,1,1,1,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

The input file of Weka is in arff format[11]. Above is an instance of one application in arff format. The input data is in 33 dimensions. The first 14 dimensions are permission features, they are shown in Table II in order. The next 8 dimensions are PhoneGap JavaScript functions shown in Table III orderly. The last 11 dimensions are unsafe JavaScript functions, they are "document.write()", "document.writeln()", "innerHTML()", "outerHTML()", "html()", "append()", "prepend()", "before()", "after()", "replaceAll()", "replaceWith()".

B. Training, Predicting and Result Analysis

The experimental data comes from[5]. We get a list of normal apps and a list of vulnerable apps by sending emails to authors of [5]. Then we download apps from google play according to the list. Finally, we get 578 normal apps and 408 vulnerable apps. In our experimental data, 300 normal apps and 300 vulnerable apps are used for training and other 278 normal apps and 108 vulnerable apps are used for detection.

There are many indicators to evaluate whether an detection algorithm is good or bad, such as *True Positive Rate(TPR)*, *False Positive Rate(FPR)*, *Precision*, *F-Score(F-Measure)* and *Overall Accuracy(ACC)*. Let *TP* denote the number of malicious apps that are classified as malicious apps, *FN* represent the number of malicious apps that are classified as normal apps. On the contrary, *TN* refers to the number of normal apps that are classified as normal apps, *FP* represents the number of normal apps that are classified as malicious apps. Then the definition of *TPR* is

$$TPR = \frac{TP}{TP + FN} \tag{1}$$

The definition of *FPR* is

$$FPR = \frac{FP}{FP + TN} \tag{2}$$

The definition of *Precision* is

$$Precision = \frac{TP}{TP + FP} \tag{3}$$

The definition of *F-Score* is

$$F - Score = \frac{2 \times Precision \times TPR}{Precision + TPR} \tag{4}$$

The definition of *ACC*[12] is

$$ACC = \frac{TP + TN}{TP + TN + FP + FN} \tag{5}$$

When *TPR* is higher and *FPR* is smaller, the detection system will achieve a better performance. *Precision* assesses the predictive power of the algorithm. *F-Score* is used to evaluate the detection system synthetically. *ACC* estimates the overall effectiveness of the algorithm[13].

Machine learning has been a hot topic, many researchers solve problems using machine learning algorithms[14,15]. We employ 9 classification algorithms to do detection, including BayesNet, NativeBayes, LibSVM, SMO, IBk, J48, RadomForest, RandomTree and DecisionTable. These names are all classes of Weka framework. They are a little bit different from terms of machine learning. BayesNet is Bayes Network in machine learning, it is a probabilistic graphical model (a type of statistical model) that represents a set of random variables and their conditional dependencies via a directed acyclic graph (DAG). NativeBayes classifiers are a family of simple probabilistic classifiers based on applying Bayes' theorem with strong (naive) independence assumptions between the features. LibSVM is similar to support vector machines (SVM). SVM are supervised learning models with associated learning algorithms that analyze data and recognize patterns. SMO is short for sequential minimal optimization, it is an algorithm for solving the quadratic programming (QP) problem that arises during the training of support vector machines. IBk is the same as the k-Nearest Neighbors algorithm (k-NN), it is a non-parametric method used for classification. J48 generates a pruned or unpruned C4.5 decision tree. RandomTree is a tree or arborescence that is formed by a stochastic process. RadomForest constructs a forest of random trees. Decision tables are used to model complex rule sets and their corresponding actions. We summarize the output of Weka, and get the result of Table IV.

As we mentioned above, as for a good classification algorithm, its *TPR*, *Precision*, *F-score* and *ACC* should be high, and its *FPR* should be low. From Table IV, We can see that RandomForest is the best classification method. The *TPR* of RandomForest is 95.3% and the *FPR* is 8%. RandomTree is the second best classification method, of which the *TPR* is 94.6%, and the *FPR* is 8.3%.

TABLE IV. POPULAR PHONEGAP PLUGINS

Algorithm	***TPR***	***FPR***	***Precision***	***F-Score***	***ACC***
BayesNet	0.943	0.090	0.943	0.943	94.3005%
NaiveBayes	0.943	0.090	0.943	0.943	94.3005%
LibSVM	0.946	0.089	0.945	0.945	94.5596%
SMO	0.935	0.082	0.936	0.936	93.5233%
IBk	0.935	0.082	0.936	0.936	93.5233%
J48	0.938	0.081	0.939	0.938	93.7824%
RandomForest	0.953	0.080	0.953	0.953	95.3368%
RandomTree	0.946	0.083	0.945	0.946	94.5596%
DecisionTable	0.940	0.080	0.941	0.941	94.0415%

The time of prediction is no more than 1 second. Plus the preprocessing time, the average time to recognize one application is no more than 2 seconds. As for the detection method proposed in [5], the average running time for each app is 15.38 seconds. Our method is faster. The method in [5] used a java library called WALA to find function call sequences to do static analysis. The time complexity of the algorithm in [5] is very high, approximately $O(n^4)$. While in our detection method, we do not try to find function call sequences. Instead we use string match functions in Python to judge whether an function is used in the specific application.

The detection result is examined by hand in [5]. This is a huge amount of work. But in our method, we can easily get FPR automatically. The precision of the detection method in [5] is 97.7%, while our best result can also achieve 95.3%, only two percent lower. But we improved a lot in saving time.

IV. Prevention of Code Injection Attacks

There are many ways to prevent vulnerable applications from code injection attacks. We can prevent in the mobile operating system, in the PhoneGap framework, in the application level.

A. Improved Fine-Grained Access-Control Model

To prevent code injection attacks, we improve an existing fine-grained access-control model[16] to control the abuse of permissions. An HTML page or a nested frame should not be given the whole permissions of an application. In one HTML page, there may be two or more frames from different origins. Hence the operating system needs to identify which origin is trustful, and then gives specific permissions to the specific HTML page or the specific nested frame.

Iframe is usually utilized to display advertisements in HTML pages. There may be some attacks in advertisement exhibition. The existing access control model is mainly used to prevent attacks from displaying advertisement. It only restricts permissions in the frame, not in the HTML page. In this paper, we also restrict permissions in the HTML page. We improve the existing access-control model, so that this model not only allows developers to assign specific permissions to the specific nested frame but also to the specific HTML page. Examples of HTML permission declaration are shown in Table V.

TABLE V. HTML Declaration

Attribute Value	**Examples**
Not empty	HTML: <head permission="READ_CONTACT" src="http://www.example.com">
Empty	HTML: <head permission="" src="http://www.example.com">
NULL	HTML: <head permission="NULL" src="http://www.example.com">
Not specified	HTML: <head src=http://www.example.com>

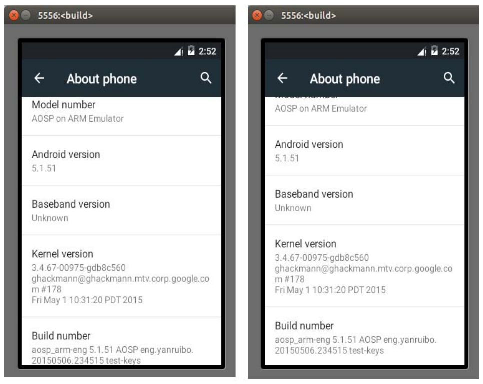
Through this access control model, the specific HTML page owns permissions it should have. This can solve the amplification and abuse of permissions. In this way, the HTML page does not have permissions that the malicious code needs. Even if malicious code is injected into a specific HTML page, the malicious code injected will not be executed successfully.

B. Adding Filters

The idea of adding filters is to filter data that read from external channels to remove injected JavaScript code. There are many places to add filters. We can choose to add filters in the PhoneGap Native library, or in the framework level of Android source code. Adding filters in the PhoneGap library is achieved in [5]. In this paper, we choose to add filters in the third level of Android source code, the framework level.

Adding filters in the Android framework benefits all the middelwares such as PhoneGap, Appcelerator etc. Because the injected code is removed in Android OS level, applications wrote by any middleware cannot suffer from code injection attacks. But there are disadvantages. It needs to modify many places of Android source code, since there are many channels, such as contacts, SMS, Wi-Fi, Bluetooth and so on. Call functions of these channels are not the same. Every channel has its own call functions. So we need to modify code of all the channels. This is a huge amount of labor. In this paper, we only modify one channel, contacts. We modify the framework of Android source code.

We download the latest edition of Android source code[17]. To remove injected code, we use an existing java library, jsoup[18]. In jsoup library, there is a method called "clean()" which was used before to solve the problem of XSS. It can also be used to solve code injection problems. The method "clean()" is called in function "getString()" of class "android.database.CursorWrapper" to remove injected code in contacts. In order to make Android system run smoothly, a new java thread is established to do clean operation. We use Android 5.1.51 edition, aosp_arm-eng compile type. Fig. 3 shows the detailed information of Android edition complied by us.



(a)

Model number
AOSP on ARM Emulator

Android version
5.1.51

Baseband version
Unknown

Kernel version
3.4.67-00975-gdb8c560
ghackmann@ghackmann.mtv.corp.google.co
m #178
Fri May 1 10:31:20 PDT 2015

Build number
aosp_arm-eng 5.1.51 AOSP eng.yanruibo.
20150506.234515 test-keys

(b)

AOSP on ARM Emulator

Android version
5.1.51

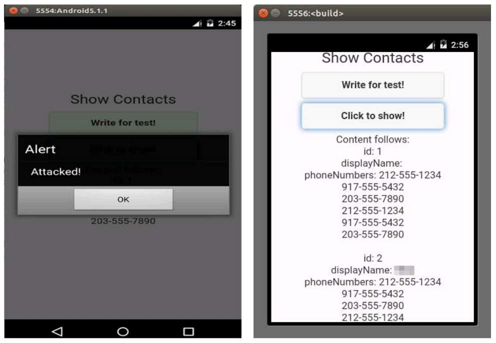
Baseband version
Unknown

Kernel version
3.4.67-00975-gdb8c560
ghackmann@ghackmann.mtv.corp.google.co
m #178
Fri May 1 10:31:20 PDT 2015

Build number
aosp_arm-eng 5.1.51 AOSP eng.yanruibo.
20150506.234515 test-keys

Fig. 3. Android Edition Information

We run ShowContacts application on our modified Android OS. The experimental result shows that the code injection attacks are prevented. Fig. 4 (a) shows the code injection attacks on normal Android OS, and Fig. 4 (b) shows the code injection attacks are prevented on our modified Android OS.



(a)

(b)

Content follows:
id: 1
displayName:
phoneNumbers: 212-555-1234
917-555-5432
203-555-7890
212-555-1234
917-555-5432
203-555-7890

id: 2
displayName: [redacted]
phoneNumbers: 212-555-1234
917-555-5432
203-555-7890
212-555-1234

Fig. 4. Comparation of Code Injection Attack and its Prevention

V. Related Works

In this section, we discuss the research work in XSS attacks and code injection attacks. Code injection attacks is very similar to XSS. So research on XSS attacks is useful to our work. XSS attacks happen in PC web applications while code injection attacks happen in mobile applications. There are many code injection attack channels on mobile devices, such as scanning barcode, connecting Wi-Fi and playing multi-media files.

XSS Attacks. Many researchers have done a lot of work on XSS detection. One is based on taint analysis, such as the automated system to detect and validate DOM-based XSS vulnerabilities[19], the method called ACTARUS[20] and the tool named Flowdroid[21]. There are also many other methods for XSS detection. For example, data mining methods to predict SQL injection and XSS in web applications[22] and prediction models on both classification and clustering to predict vulnerabilities[23] are based on machine learning. Fabien Duchene et al. presented a way to detect web injection vulnerabilities by generating test inputs with a combination of model inference and evolutionary fuzzing[24]. In addition, there are many approaches to preventing XSS. One is through sanitization. Noxes is put forward to mitigate XSS attacks in [25]. ScriptGard and XSS-GUARD are proposed to prevent XSS attacks in [26,27]. SWAP (Secure Web Application Proxy)[28] and xJS[29] are raised to prevent code injections in web applications. Philipp Vogt et al. proposed a solution to stop XSS attacks on the client side by tracking the flow of sensitive information inside the web browser[30].

Code Injection Attacks. Since code injection attacks are first presented in 2014[4], there are a few research works on code injection attacks. Xing Jin et al. proposed code injection attacks in 2014[4], but the injected JavaScript code is in plaintext. So the attack may be recognized by application users. If JavaScript code is injected in Wi-Fi SSID or in a filename, users can see the injected JavaScript code and will not be cheated. Furthermore, Xing Jin et al. proposed detection method and solutions in [5]. They used a tool called WALA to do static analysis on Android source files, and employed a specific algorithm to find JavaScript call function sequences to detect vulnerable applications. In the application level, they advised developers should use unsafe JavaScript functions as little as possible, although unsafe JavaScript functions may be unavoidable. In the middleware framework level, Xing Jin et al. have implemented a prototype named NoInjection as a patch to PhoneGap in Android to defend against the attack. In the operating system level, the implementation of WebView can be modified to make the browser to ignore the injected JavaScript code automatically. To do this, CSP(Content Security Policy) strategy can be adopted[31].

VI. Conclusion

In this paper, we introduce a new type of code injection attacks, coding-based code injection attacks, which are more hidden. We also present a new detection method based on classification algorithms of machine learning. Our methods run fast and the *Precision* can reach at 95.3% in our best classification method. Finally, we improve an existing access-control model and add filters in the Android framework level to prevent code injection attacks.

Acknowledgment

We would like to thank the anonymous reviewers for their helpful advice. We would like to thank the authors of [5] to provide the experimental data for us. We also thank Chunhui Chen for the helpful discussion.

References

[1] Xi Xiao, Zhenlong Wang, Qing Li, et al. “ANNs on Co-occurrence Matrices for Mobile Malware Detection”, The KSII Transactions on Internet and Information Systems, vol.9, no.7, pp.2736-2754.

[2] Xi Xiao, Xianni Xiao, Yong Jiang, et al. “Detecting Mobile Malware with TMSVM” Proceeding of the SECURECOMM 2014, 2014.

[3] HTML5. http://en.wikipedia.org/wiki/HTML5.

[4] X. Jin, T. Luo, D. Tsui, W. Du. Code injection attacks on HTML5-based mobile apps. In MoST, 2014.

[5] Jin X, Hu X, Ying K, et al. "Code injection attacks on HTML5-based mobile apps: Characterization, detection and mitigation." Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2014, pp.66-67.

[6] PhoneGap, http://phonegap.com.

[7] XSS, https://en.wikipedia.org/wiki/Cross-site_scripting.

[8] WebView,https://developer.android.com/reference/android/webkit/WebView.html.

[9] Weka, http://www.cs.waikato.ac.nz/ml/weka.

[10] Xiaoyan, Zhao, Fang Juan, Wang Xiujuan. “An Android malware detection based on permissions.” 2014 International Conference on Information and Communications Technologies (ICT 2014), Nanjing, China, May 2014, pp.2.063-2.063.

[11] ARFF, https://weka.wikispaces.com/ARFF.

[12] Ham H S, Kim H H, Kim M S, et al. "Linear SVM-based android malware detection." Frontier and Innovation in Future Computing and Communications. Springer Netherlands, 2014, pp.575-585..

[13] Sokolova, Marina, Nathalie Japkowicz, et al. "Beyond accuracy, F-score and ROC: a family of discriminant measures for performance evaluation." AI 2006: Advances in Artificial Intelligence. Springer Berlin Heidelberg, 2006, pp.1015-1021.

[14] Gao N, Gao L, Gao Q, et al. "An Intrusion Detection Model Based on Deep Belief Networks." Advanced Cloud and Big Data (CBD), 2014 Second International Conference on. IEEE, 2014, pp.247-252.

[15] Chen Z, Wu D, Xie W, et al. "A Bloom Filter-Based Approach for Efficient Mapreduce Query Processing on Ordered Datasets." Advanced Cloud and Big Data (CBD), 2013 International Conference on. IEEE, 2013, pp.93-98.

[16] Jin X, Wang L, Luo T, et al. "Fine-grained access control for html5-based mobile applications in android." Proceedings of the 16th Information Security Conference (ISC). 2013.

[17] Android Source, https://source.android.com/source/downloading.html.

[18] Jsoup: Java HTML Parser, http://jsoup.org/.

[19] Lekies, Sebastian, Ben Stock, et al. "25 million flows later: large-scale detection of DOM-based XSS." Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. ACM, 2013, pp.1193-1204.

[20] Guarnieri S, Pistoia M, Tripp O, et al. "Saving the world wide web from vulnerable JavaScript." Proceedings of the 2011 International Symposium on Software Testing and Analysis. ACM, 2011, pp.177-187.

[21] Arzt S, Rasthofer S, Fritz C, et al. "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps." ACM SIGPLAN Notices. ACM. Vol. 49, No. 6,2014, pp.259-269.

[22] Shar, Lwin Khin, Hee Beng Kuan Tan. "Mining input sanitization patterns for predicting SQL injection and cross site scripting vulnerabilities." Proceedings of the 34th international conference on software engineering. IEEE Press, 2012., pp.1293-1296.

[23] Shar, Lwin Khin, Hee Beng Kuan Tan, Lionel C. Briand. "Mining SQL injection and cross site scripting vulnerabilities using hybrid program analysis." Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, 2013, pp.642-651.

[24] Duchene F, Groz R, Rawat S, et al. "XSS vulnerability detection using model inference assisted evolutionary fuzzing." SECTEST 2012-3rd International Workshop on Security Testing (affiliated with ICST). IEEE Computer Society, 2012, pp.815-817.

[25] Kirda E, Kruegel C, Vigna G, et al. "Noxes: a client-side solution for mitigating cross-site scripting attacks." Proceedings of the 2006 ACM symposium on Applied computing. ACM, 2006, pp.330-337.

[26] Saxena, Prateek, David Molnar, et al. "SCRIPTGARD: automatic context-sensitive sanitization for large-scale legacy web applications." Proceedings of the 18th ACM conference on Computer and communications security. ACM, 2011, pp.601-614.

[27] Bisht, Prithvi, V. N. Venkatakrishnan. "XSS-GUARD: precise dynamic prevention of cross-site scripting attacks." Detection of Intrusions and Malware, and Vulnerability Assessment. Springer Berlin Heidelberg, 2008, pp.23-43.

[28] Wurzinger P, Platzer C, Ludl C, et al. "SWAP: Mitigating XSS attacks using a reverse proxy." Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems. IEEE Computer Society, 2009, pp.33-39.

[29] Athanasopoulos E, Pappas V, Krithinakis A, et al. "xJS: practical XSS prevention for web application development." Proceedings of the 2010 USENIX conference on Web application development. USENIX Association, 2010.

[30] Vogt P, Nentwich F, Jovanovic N, et al. "Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis." NDSS. 2007.

[31] Stamm, Sid, Brandon Sterne, Gervase Markham. "Reining in the web with content security policy." Proceedings of the 19th international conference on World wide web. ACM, 2010, pp.921-930.