Secure HybridApp: A Detection Method on the Risk of Privacy Leakage in HTML5 Hybrid Applications Based on Dynamic Taint Tracking

Degang Sun, Chenyang Guo, Dali Zhu, Weimiao Feng
Institute of Information Engineering
Chinese Academy of Sciences
Beijing, China
e-mail:{sundegang, guochenyang, zhudali, fengweimiao}@iie.ac.cn

***Abstract*—In the past few years, HTML5-based mobile applications are becoming more and more popular because they can run across different platforms, which greatly reduces the developing cost and improves the production efficiency. This kind of app is also called hybrid app. It can access the platform resources mostly like a native app with the help of many third-party frameworks. As we all know, web apps are prone to many kinds of attacks which can cause privacy leakage. HTML5 apps are a kind of web app, which means they can also be attacked by these web attacking methods. Due to the dynamic nature of hybrid apps, it is very hard to analyze the malicious behavior in them based on static or dynamic code analysis. But dynamic taint tracking method is very suitable for this task, it treats the user privacy as taint data and check whether it will be leaked out through illegal channels. Once this kind of action is found, we can stop it immediately and notice the app user about it.**

In this paper, we mainly talk about the privacy data, the privacy leakage channels in HTML5 hybrid apps. And we propose a dynamic method to avoid privacy leakage based on dynamic taint tracking in Android. It can be applied to other systems.

Keywords-HTML5-based applications, privacy leakage, dynamic taint tracking, detecting method

I. INTRODUCTION

A. HTML5-Based Applications

HTML5-based applications are developed with the most commonly used technologies like HTML5, JavaScript, CSS3. These technologies are the basis of the Internet and very easy to learn. So this developing method can greatly improve the production efficiency for applications [1].

They run in browsers and web containers, which can render HTML files and execute JavaScript programs. Today, all mobile devices access the Web to communicate with the world, so almost all platforms have a web container, e.g., WebView in Android, UIWebView in iOS and WindowsBrowser in Windows Phone. This nature gives these applications the ability to run across different platforms. And this development method can achieve the goal "write once, run anywhere" [2].

In order to provide system functionalities, e.g., accessing Camera and GPS, for the web content, the OS builds a bridge between the web content and native resources, e.g., "*addJavaScriptInterface()*" in Android. This bridge can bind a native object directly to a JavaScript object. The JavaScript functions can use the object to access native resources and vice versa. Thus the HTML5-based applications have almost the same ability as native apps [3].

However, HTML5-based applications are still web apps. Similar to traditional web apps, they are vulnerable to code-injection attacks. What's worse, every page including every iframe in the app shares all the privileges[4][5]. So if any malicious code is injected into one of the pages in the app, it can access all the resources that the app is allowed to use. It is not that dangerous if the malicious code can only access the private data. What makes it horrible is that there are many ways for the attackers to send the data out. For example, the traditional way HTTP request, the new born HTML5 postMessage method and the WebSocket protocol[6]. Once these channels are misused, they are good channels for the attackers to deliver your privacy out. Thus, it is very necessary to propose a way to detect the data leakage risks and protect user data.

We implemented our detecting method in a tool called SecureHybridApp based on dynamic taint tracking in Android. In our system, we will taint the privacy data as tainted sources and track it to make sure whether it sinks in one of the privacy leakage channels. And once it happens, we will prompt it to the app user to notice him about the data leakage.

II. PRIVACY LEAKAGE ANALYSIS

A. Privacy Leakage Channels

1) Traditional HTTP methods

TABLE I. AN EXAMPLE OF GEOLOCATION LEAKAGE

```
<img src=x onerror=plus.geolocation.watchPosition(
function(loc){
    b=document.createElement("img");
    b.src=http://evil.com/loc?lon=
        +loc.coords.longitude+"&lat="
        +loc.coords.latitude});
/>'"
```

First, traditional HTTP method can be a leakage channel. The HTTP Get and Post are the mostly used method for

978-1-4673-9026-2/16/$31.00 ©2016 IEEE

applications to communicate with the server-side. In Jin et al's code injection method, they inject an "*IMG*" tag with error into the app page in an HTML5 hybrid app [1], which triggers the "*onerror*" method of the tag. They put some malicious code into that method, which firstly calls the Geolocation API to get the location information. Then it uses the HTTP Get method to send the information to an evil target, "*evil.com*". The code is shown below in Table I.

2) Websocket protocol

Second, WebSocket protocol can be a good channel for attackers. To improve the ability of web applications, HTML5 defines many new features. WebSocket protocol is one of them which enables applications to maintain bidirectional communications with server-side processes. In Android 5.0, the WebView begins to support websocket. This protocol is very useful in real-time applications that need frequent data exchange with the server-side and it sharply reduces the cost caused by former methods like ajax polling and long polling. However, every coin has two sides, it also provides a good channel for attackers to transfer data[6].

3) HTML5 postmessage method

Third, postMessage method is another feature of HTML5 which safely enables cross-origin communication. Normally, scripts on different pages are allowed to access each other if and only if the pages that executed them are at locations with the same protocol (usually both https), port number (443 being the default for https), and host. postMessage provides a controlled mechanism to circumvent this restriction in a way which is secure when properly used [7][8]. However, it can become a good channel for attackers if it is misused.

4) API Data Transfer Channels

The last but not the least, because hybrid apps can use platform APIs provided by many third-party frameworks, it can use the API related function, e.g., sending messages to send out sensitive data.

B. Privacy Data

There are mainly 3 kinds of privacy data in the HTML5 apps, cookie, HTML5 offline storage and data got from the Geolocation API and other platform APIs.

1) Cookie

Cookie is generated from the server-side, then sent to the client-side (usually a browser) and saved in temp files. It contains the session data, user account info and authentication info and so on. The server-side uses these data to verify users and identify their status. These data are generally user's private data. If attackers steal these data, they can use your account info to login the websites and read your account data, they can get your phone number and email address and so on. So when we use cookie, we should be careful with what website we are in and what links we click.

2) Offline storage

The volume of cookie is often small, ordinarily 4 KB, which constraints the ability of web applications. To solve this problem, the HTML5 standard proposes a new storage method called offline storage, including local storage and session storage, the content of which is much larger than that of cookies, generally 5MB. This new feature greatly improves the offline storage capability of HTML5 applications and make it possible for these apps to run without network like native ones. But the more data you put into the offline storage, it means the more private data you put in front of the Internet and the larger risk of data leakage you will take.

3) Geolocation API

In addition to the local storage strategy, HTML5 provides the Geolocation API to all web apps for the population of LBS (Location Based Service). But once it is permitted by the user, the Geolocation API can be called by any script in the page, the location data can be easily got from this API by the attacker.

4) Resource APIs

To extend the ability of HTML5 applications, many third-party frameworks provides platform APIs which allows these apps to access platform resources. Once these APIs are called by attackers, they can easily get the native user data which will cause serious privacy leakage. Thus these APIs must be treated as privacy data, so is the data got from these APIs.

III. Potential Attacking Method

No matter in traditional web sites or the new-born hybrid apps, the ads are very common to see. It is sometimes a good way for the application vendors to market their apps and a common way for free apps to make a profit. But for the users, the ads are often the source of attacks. It carries malicious code and steal the user private data through various channels.

A. Privacy Leakage from Dalvik

Some attacks come from the Java side, they get the privacy data and spread it out through HTTP or other data channels. It has been studied thoroughly in android app [9]. This paper mainly focuses on the web code which plays more important role in hybrid apps.

B. Privacy Leakage from JavaScript Engine

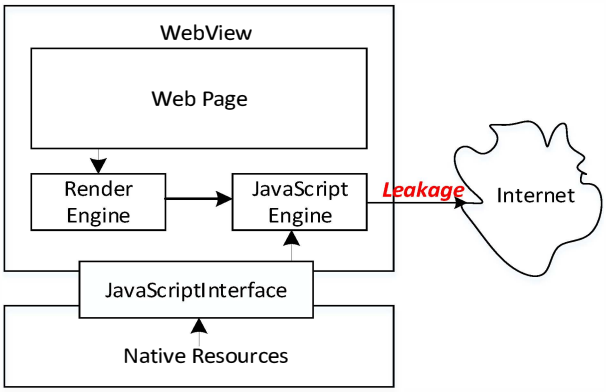

Figure 1. Data leakage modle in javascript engine

In hybrid apps, malicious code often comes from the code injection attacks and ads displayed in iframes of the page [3]. In Jin's research, there are more injection channels

in hybrid apps than the traditional web apps because there are more data input channels, e.g., barcode input, Wi-Fi list, contacts and so on [1]. So the inspection work is more complicated than before. But the data leakage channels remain limited. And the leakage operation is finally happened in JavaScript engine. So it is easier to find data leakage in the JavaScript engine. Fig. 1 shows the most common data leakage flow in hybrid apps.

IV. DYNAMIC TAINT TRACKING

Dynamic taint tracking is a classic method in finding software vulnerabilities and tracking data leakage procedures [10]. This approach does not need source code or special compilation for the monitored application. It mainly focuses on the sensitive data origin and the potential leakage sinks. As we analyzed above, the privacy data is widely distributed in HTML5 hybrid app. And due to the dynamic nature of the web app and especially the JavaScript language, it is very complicated to analyze its call procedure [11][12]. It is easier to find privacy leakage phenomenon based on dynamic taint tracking method. According to the situation we are in, it is appropriate to use this method. Also it can be used to accomplish fine-grained privacy leakage detection [9], [11].

A. Tainted Sources

We take the privacy data as tainted sources. Although Geolocation permission control is realized by Android system, once the app user permitted the request, every script in the page can access the Geolocation API, including the potential malicious codes in iframe. So in order to avoid attacks from iframes, we should add an extra protection to it in our system. Other sensitive functional APIs like accessing the camera, sensors are not likely to cause data leakage. So we mainly focus on some data accessing APIs like fetching user contacts, call logs, device IMEI and the Geolocation data. We change the data generation code to add tainted data. The sources are listed below in Table II.

TABLE II. PRIVACY DATA LIST AND TAINT FLAGS

Private data	Tainted flag	Data type
Cookies	0x00000002	String
Offline storage	0x00000003	String
Contacts	0x00000004	String
Call logs	0x00000005	String
SMS	0x00000006	String
IMEI	0x00000007	String
IMSI	0x00000008	String
Location data	0x00000009	String

1) Tainted cookies

Cookies are operated in *CookieManager* mainly with "*SetCookie*", "*SetCookieSync*" and "*GetCookie*" methods. We shouldn't taint the origin data, instead we taint the data copies when they are accessed with the method "*GetCookie*". This method is generally called in JavaScript engine, so we should add an attribute "taint", the value of which is the flag "0x00000002", to the return String object.

2) Tainted offline storage

Local storage and session storage are operated with "*getItem*" and "*setItem*" methods. We tainted the offline storage data when they are read with the method "*getItem*". We also taint these data with its flag "0x00000003".

3) Tainted geolocation data

We add tainted flag to Geolocation data in the *latitude()* and *longitude()* method, which will return the tainted data.

4) Tainted native rsources

Native resources are brought to the web pages through the Java-to-JavaScript bridge. So if we want to track the usage of native resources, we have to taint the native resources we focus on, e.g., the contacts, call logs, SMS, IMEI and so on.

Similarly, we tainted these sensitive data when they are accessed with their "get" method.

In Dalvik, the structure of tainted String object is like figure 2, it points to an array object. We put the taint flag in the array object.

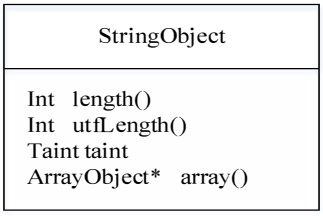

Figure 2. Tainted string object in dalvik

B. Taint Propagation

To make sure that the sensitive data can't be washed away by some data transforming operations and data copies be tainted in the same way. We should do something in the propagation procedure.

There are totally two transferring paths of the tainted data. The first one is about cookie and the offline storage, they are stored in the Chromium at the beginning. So its path is simply from Chromium render engine to JavaScript engine. The other is about native resources. These data are stored in Android platform and got through the "bridge". So the path is first from Dalvik to Chromium and then from Chromium to JavaScript engine.

1) Propagation form dalvik to chromium

This is about data transferring from Dalvik to Chromium. In browsers, it is often necessary to allow JavaScript to interact with browser plugins. This standard about browser plugins was called Netscape Plugin Application Programming Interface (NPAPI), which provides a cross-platform architecture for browsers in WebKit, and now it is still in use in Chromium, which is put into use since Android 4.4. It allows JavaScript within a browser to access the APIs of plugins, and vice versa. In Android, from the WebView's perspective, Java is treated just as a plugin, and invocation of Java code from JavaScript follows the NPAPI standard.

In the Java implementation of NPAPI standard, an instance "*javaFoo*" of a Java class called "*ClassFoo*" is bound to WebView through *addJavaScriptInterface()*, resulting to a new JavaScript object called "*jsFoo*" in WebView. When the JavaScript code in WebView invokes *jsFoo.bar()*, a series of actions will be performed, leading to

the eventual invocation of the bar method of the Java object *javaFoo*.[1]

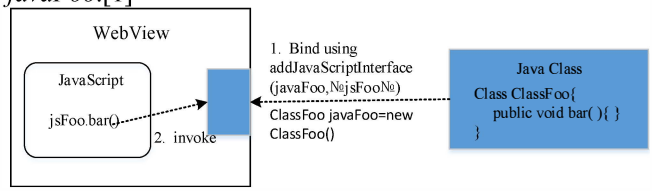

Figure 3. Java to javascript binding process

Fig. 3 shows the type transformations in the invocation process.

We add taint flags to *JavaValue* and *NPVariant* structures and we add taint tracking operations to converting functions *JavaValueToNPVariant* and *NPVariantToV8Object*. To get taint flags, we add *Taint()* method to V8 String object. Fig. 4 shows the data type change in Chromium.

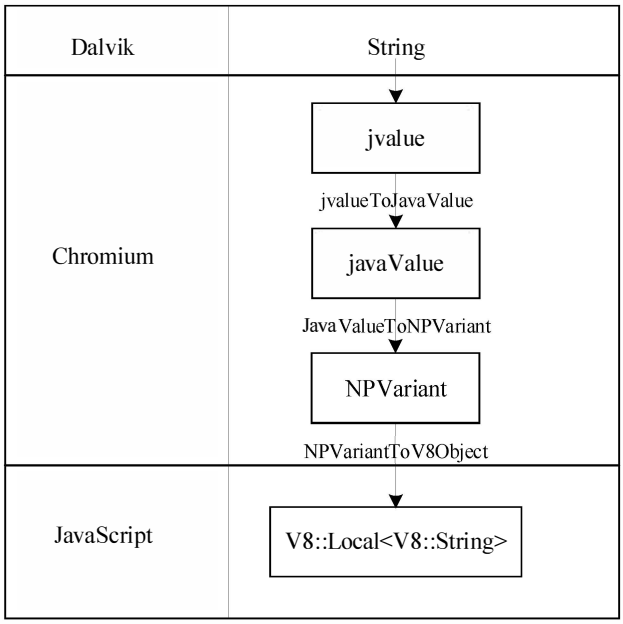

Figure 4. The data type change in chromium

2) Propagation from Chromium to JavaScript Engine

In Chromium render engine, the Java instance exists as a globally accessible variant under the DOM tree. We can access it with *JsFoo* like in Fig. 3 in every JavaScript function, and the data type of the return from this call will be transferred into a V8 (the JavaScript engine used in Chromum) String.

3) Propagation in JavaScript Engine

In JavaScript engine, the String object will be operated with some JavaScript functions like substring, assignment, join and so on. We add taint propagation operations in these functions to maintain the taint flag. The related taint operation rules are like table III.

TABLE III. DATA PROPAGATION RULE IN JAVASCRIPT ENGINE

Operations	Meaning	Propagation rule	Description
A=B	VA ←VB	T(VA) ←T(VB)	Taint when assigning
A = substr(B,i,j)	VA←VB	T(VA) ←T(VB)	Taint when substr
A=B+C	VA← VB+ VC	T(VA)← T(VB)+ T(VC)	Taint the new object when adding

C. *Taint Checking*

We check the taint data in the potential data leakage channels. These protocols are realized in Chromium web stack, e.g., HTTP, XMLHttpRequest, postMessage and WebSocket. We analyze the data sending through these channels. If any tainted data are found, we will check the tainted flag table to identify what kind of data it is and check whether the target origin is legal. If it is illegal, we will prompt an alert to the user.

The place we put the check logic in these channels are like table IV.

TABLE IV. TAINTING CHECKING METHODS

Target	Method
HTTP	URLRequestHttpJob
XMLHttpRequest	XMLHttpRequest.cc send()
Websocket	WebSocket.cc SendMessage()
postMessage	webmessageportchannel_impl.cc postMessage()

V. EXPERIMENT

We test our method in a hybrid app called html5+ in Android 5.1.1. We put the malicious code in the iframe of the main page and we trick the user with a popular “2048” game like figure 5. The malicious code in the frame is like table V. If the user clicks the “play” button in it. The malicious code will be triggered and send the IMEI out to “xxx.xxx.com” through websocket. At the same time, the SecureHybridApp system will track the tainted message. If data leakage is found in the above leakage channels. It will prompt to the user like figure 6. As we experimented, the SecureHybridApp system successfully checked out the data leakage channel in an ad hidden in the iframe in the page.

TABLE V. MALICIOUS CODE IN THE IFRAME

```
function getDeviceInfo(){
    var socket = new WebSocket('ws://xxx.xxx.com/');
    socket.onopen = function(event) {
        socket.send(plus.device.imei); //The device IMEI
    };
}
```

We also evaluate the performance impact of our work on applications. We mainly focus on the applications that attach APIs to WebView using *addJavascriptInterface()*. We measure the followings: performance cost in Dalvik and in JavaScript engine. the result tells the performance cost in Dalvik is about 22% and the performance cost in JavaScript engine is 4.3%.

Figure 5. The app html5+

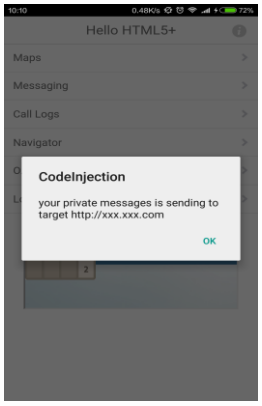

Figure 6. The SecureHybridApp prompt

VI. CONCLUSION

HTML5 Hybrid apps are becoming more and more popular these years. At the same time, more attention must be paid to its security problems because of the nature of web app. Former methods mainly focus on the static or half dynamic analysis of these apps. These methods can draw out the call graph of the app, but this procedure takes a lot of time or can only be done statically. Dynamic taint tracking makes it possible to track the privacy data and make sure the privacy data not to be leaked out by malicious code. SecureHybridApp is based on this method and it can dynamically detect the privacy data leakage action.

REFERENCES

[1] Jin, Xing, et al. "Code Injection Attacks on HTML5-based Mobile Apps: Characterization, Detection and Mitigation." Acm Sigsac Conference on Computer & Communications Security ACM, 2014:66-77.

[2] Jin, Xing, et al. Fine-Grained Access Control for HTML5-Based Mobile Applications in Android. Information Security. 2015.

[3] Georgiev M, Jana S, Shmatikov V. Breaking and Fixing Origin-Based Access Control in Hybrid Web/Mobile Application Frameworks.[J]. Ndss Symposium, 2014, 2014:1-15.

[4] Akhawe D, Saxena P, Song D. Privilege separation in HTML5 applications[C]// Usenix Conference on Security Symposium. USENIX Association, 2012:23-23..

[5] Luo T, Hao H, Du W, et al. Attacks on WebView in the Android system[C]// Twenty-Seventh Computer Security Applications Conference, ACSAC 2011, Orlando, Fl, Usa, 5-9 December. 2011:343-352.

[6] Wiki. WebSocket. https://en.wikipedia.org/wiki/WebSocket.

[7] Dong G, Zhang Y, Wang X, et al. Detecting cross site scripting vulnerabilities introduced by HTML5[C]// Computer Science and Software Engineering (JCSSE), 2014 11th International Joint Conference on. IEEE, 2014:319-323.

[8] Yu J, Yamauchi T. Access Control to Prevent Attacks Exploiting Vulnerabilities of WebView in Android OS[C]// IEEE International Conference on High PERFORMANCE Computing and Communications. 2013:1628-1633.

[9] Enck, William, et al. "TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones." Usenix Conference on Operating Systems Design & Implementation 2015:393-407.

[10] Song, By J Newsomeandd. "Dynamictaintanalysisforautomaticdetection,analysis,andsignaturegeneration ofexploitsoncommodity software.In NDSS,2005.".

[11] Brucker, Achim D., and M. Herzberg. On the Static Analysis of Hybrid Mobile Apps. Engineering Secure Software and Systems. Springer International Publishing, 2016.

[12] Chen, Yen Lin, H. M. Lee, and A. B. Jeng. "DroidCIA: A Novel Detection Method of Code Injection Attacks on HTML5-Based Mobile Apps." IEEE Trustcom/bigdatase/ispa IEEE Computer Society, 2015.

[13] Vogt, Philipp, et al. "Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. " Network and Distributed System Security Symposium, NDSS 2007, San Diego, California, Usa, February -, March 2007.