LETTER *Special Section on Information and Communication System Security—Against Cyberattacks—*

Access Control to Prevent Malicious JavaScript Code Exploiting Vulnerabilities of WebView in Android OS*

Jing YU[†], *Nonmember and* **Toshihiro YAMAUCHI**[†a)], *Member*

SUMMARY Android applications that using WebView can load and display web pages. Interaction with web pages allows JavaScript code within the web pages to access resources on the Android device by using the Java object, which is registered into WebView. If this WebView feature were exploited by an attacker, JavaScript code could be used to launch attacks, such as stealing from or tampering personal information in the device. To address these threats, we propose an access control on the security-sensitive APIs at the Java object level. The proposed access control uses static analysis to identify these security-sensitive APIs, detects threats at runtime, and notifies the user if threats are detected, thereby preventing attacks from web pages.

key words: *Android, WebView, static analysis, access control*

1. Introduction

In the last several years, the Android [2] has become more popular. Android provides rich libraries, such as OpenGL, SQLite, WebKit, etc. In this paper, we focus on WebKit. WebView [3], provided by WebKit, could be used to implement a simple browser function in an Android application (hereafter, Android app), so that users can load web pages in the Android app directly without using a browser. In addition to displaying web pages, WebView allows JavaScript within web pages to invoke methods defined in the Android apps. However, if the rich features of WebView are not used properly, devices could become vulnerable to malicious attacks [4], such as those that steal personal information or tamper with data on the Android device.

Reference [5] has reported attacks using the vulnerabilities of WebView in Android, and these attacks fall into two types: attacks from web pages to the Android OS and attacks from Android apps to web pages. Reference [6] was first to recognize the threat of JavaScript code that abuse Android permissions, and the authors proposed a static analysis method to estimate the potential threat of Android apps that use WebView. However, no effective countermeasures were discussed in existing works.

In this paper, to address the attacks using the vulnerabilities of WebView from web pages, we propose an access control on the security-sensitive APIs at the Java object level. The Java object is an interface to the web pages loaded in WebView. By using the Java object, JavaScript code in the web pages can access the resources on the Android device. To detect potential attacks that use the Java object, we performed static analysis to determine the security-sensitive APIs that could be invoked by the Java object, and we ran a threat-detection process each time the Java object is registered into WebView. If a threat is found, the user is warned, and the user decides whether to allow the registration of the Java object or to disable it to prevent attacks from web pages. Because the proposed method performs access control at the Java object level, the user disables only the Java object. The Android app could continue running without enabling the suspicious Java object, and the user could browse the web page in WebView safely.

Manuscript received May 20, 2014.
Manuscript revised September 26, 2014.
Manuscript publicized December 4, 2014.

[†]The authors are with Graduate School of Natural Science and Technology, Okayama University, Okayama-shi, 700–8530 Japan.

*This paper is an extended version of the paper presented at [1].

a) E-mail: yamauchi@cs.okayama-u.ac.jp

DOI: 10.1587/transinf.2014ICL0001

2. Android OS

Dalvik is the virtual machine that runs Android apps on the Android OS. Dalvik executable files are formatted as dex (Dalvik Executable) files. An Android app written in Java is compiled and converted to a dex file, which contains all the source code of the Android app.

The permission mechanism performs access control on the resources that Android apps can access. If permissions are requested by an Android app, the user is prompted at installation time. However, the installation continues, only if the user grants all requested permissions. In addition, the permissions cannot be changed after the Android app is installed.

3. WebView

3.1 What is WebView

WebView is a component provided by a browser engine named WebKit. WebView provides basic browser functionality to load and display Web pages within Android apps without switching to the default browser. More importantly, the Android app can interact with JavaScript code embedded in web pages by using the APIs described below provided in WebView.

The setJavaScriptEnabled API enables web pages to use JavaScript, which plays an important role in the interaction. The addJavascriptInterface API registers the Java object into WebView, so that the JavaScript code within web pages could use the registered Java object to run methods defined in the Java class. The loadUrl API loads a specific

Copyright © 2015 The Institute of Electronics, Information and Communication Engineers

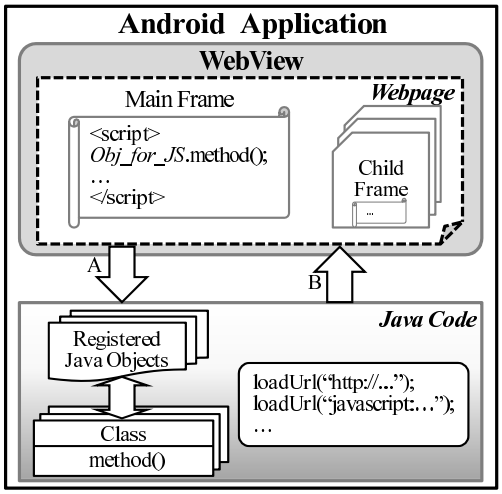


Fig. 1 Overview of Android application using WebView.

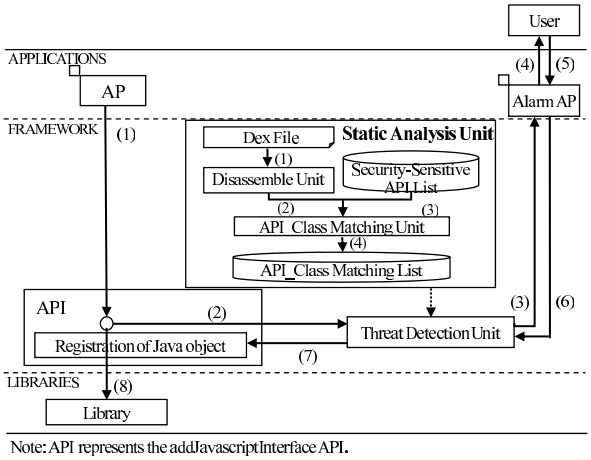


Fig. 2 Overview of proposed method.

web page.

3.2 Problems with WebView and Attack Models

Figure 1 shows an overview of an Android app that uses WebView. Path A and Path B are the two main interactions between the Android app and web pages. Path A shows that web pages loaded in WebView can invoke methods defined in the Android by using registered Java objects. Path B shows that by using the loadUrl API, an Android app can invoke JavaScript code within web pages or insert JavaScript code into web pages.

Two attack models, which also have been discussed in [5], should be considered. One model involves attacks from web pages. After the Java object is registered into WebView, all web pages loaded in WebView can use the registered Java object, regardless of the origin of the web pages. If WebView loads the malicious web pages, the JavaScript code in those malicious web pages could launch an attack, such as stealing or tampering with the personal information in the Android device. Many attack examples that exploit WebView were documented in [7].

The other model involves attacks from Android apps. Malicious Android apps can inject malicious JavaScript code into web pages to perform attacks. There are some researches against JavaScript injection attacks as Web security. These researches can be applied to web server as countermeasures of the JavaScript injection attacks. On the other hand, attacks from web pages to Android device have not really been studied. Therefore, in this paper, we propose a method to address the attacks from web pages.

4. Proposed Method

4.1 Purpose and Concept

We define threats in this paper as stealing personal information from the device and tampering and deleting personal information in the device. The threats are caused by calling security-sensitive APIs as shown in Sect. 4.4 through malicious JavaScript code. The security-sensitive APIs are defined as getting personal information from the device, sending information outside of the device, and tampering personal information of the device.

The purpose of the proposed method is to prevent malicious JavaScript code from accessing security-sensitive APIs through the Java object.

We know that the threat from JavaScript code stems from the use of security-sensitive APIs. The user is prompted to grant permissions requested by the Android app at installation time. However, a user is not aware that these permissions could be used by malicious JavaScript code in a web page to invoke the security-sensitive APIs through the Java object to attack. Therefore, we applied access control on the security-sensitive APIs at Java object level. If the security-sensitive APIs are detected in methods that the Java object could execute, we control this Java object.

One requirement for achieving this purpose is to be able to identify whether a specific Java object needs to be controlled. To achieve this requirement, the following two challenges must be met. Challenge a) is to clarify what APIs can be executed by the Java object. Challenge b) is to address the security-sensitive APIs at Java object level.

The other requirement is to be able to manage the Java object that needs to be controlled. To achieve this requirement, the following challenge must be met. Challenge c) is to prompt the user to manage the potential threat that has been addressed.

4.2 Design

Figure 2 is an overview of the proposed method, which mainly consists of three components: the Static Analysis Unit, the Threat Detection Unit, and the Alarm application. As shown in Fig. 2, our method controls the Java object at the framework layer. By intercepting the call to the addJavascriptInterface API from Android app, the information

about the Java object is sent to the Threat Detection Unit. Next, the Threat Detection Unit detects whether a potential threat exists in the Java object based on the API_Class Matching List (described in next subsection). If a threat is detected, the Alarm application is called.

The Alarm application warns the user of the threat. Then, the user replies to it to decide whether to disable the Java object. In order to support the user in making a decision, the following information is displayed: name of Android app, URL that WebView is supposed to load, name of Java object that was determined to be a threat, and security-sensitive API associated with the Java object The user can press either the "Enable" or "Disable" button located below the warning information to indicate whether to allow the use of the Java object. In addition, by clicking the URL displayed, the default browser will be invoked to load the web page. As sandbox protection is implemented in general browser, the Java object cannot be used to interact with the Android app. Therefore, if the user is not certain of the safety of the Java object, the user can use the default browser to load the web page, instead of loading it in WebView.

The Alarm application forwards the user's decision to the Threat Detection Unit. The Threat Detection Unit forwards the decision to the addJavascriptInterface API. If the user granted the use of the Java object, the Java object is registered into WebView as usual. Otherwise, the registration is aborted.

4.3 Static Analysis Unit

To determine what APIs could be executed by Java objects, we need to refer to the source code of the associated Java class. However, the Android app is compiled and packaged in the form of apk files, and we could not refer directly to the source code of the Java class. We use dexdump, which is a disassembly tool for Android, to convert the dex file into assembly code on Android OS. By analyzing the assembly code, we can determine what APIs were used. The purposes of the static analysis are to create the API_Class Matching List and to get URL that WebView is supposed to load. In order to achieve these purposes, the information of class descriptor, APIs which can be executed by class, and arguments of APIs are required for the static analysis. To make the assembly code simple and easy to be analyzed, we removed unnecessary features of dexdump excluding the three information mentioned above to make it lighter, thereby reducing the overhead of the static analysis.

Static analysis would need to be performed on every Android app at installation time. The time spent on static analysis could be perceived as part of the installation time, so the user does not feel inconvenienced because the installation time is relatively long. The static analysis must be done once on each Android app and again, if the Android app is updated.

As shown in Fig. 2, the Static Analysis Unit consists of the Disassemble Unit and the API_Class Matching Unit. The Static Analysis Unit gets the dex file from the apk file of the Android app. Next, the Disassemble Unit converts the dex format to assembly code by using dexdump. Then, the API_Class Matching Unit compares the assembly code with the Security-Sensitive API List. If the assembly code contains the the security-sensitive APIs, the Static Analysis Unit stores the API information and the associated Java class information in the API_Class Matching List.

Table 1 security-sensitive APIs.

getCellLocation	getAccounts
getDeviceId	getAuthToken
getNetworkOperator	getPassword
getPhoneType	getUserData
getSubscriberId	peekAuthToken
getLine1Number	removeAccount
getSimSerialNumber	setPassword
getVoiceMailAlphaTag	getName
getVoiceMailNumber	getProfileConnectionState
sendDataMessage	getProfileProxy
sendMultipartTextMessage	getParams
sendTextMessage	getUngzippedContent
getAllProviders	getCertificate
getBestProvider	clearHistory
getGpsStatus	clearSearches
getLastKnownLocation	getAllBookmarks
clearPassword	getAllVisitedUrls
editProperties	

4.4 Security-Sensitive APIs

In our work, we investigate on API Level 15, and we define security-sensitive APIs as the APIs that communicates with outside or that deal with personal information. Table 1 shows the list of APIs that we defined to be security-sensitive.

5. Evaluation

5.1 Experiment to Test the Operation of Proposed Method

We implemented the proposed method on Android 4.0.3. We used an Android app named HelloWebView, which has the functionality to obtain the phone number and the device ID using JavaScript code embedded in the web page and using the Java object named Obj_for_JS.

The warning information is shown in Fig. 3. By choosing "Disable", the user could deny access to the getLine1Number API and the getDeviceID API through the Java object by the proposed method.

It also confirmed that the implemented prototype of the proposed method can detect the use of all security-sensitive APIs described in Sect. 4.4 by other experiments. This results show the proposed method can detect the potential threats as mentioned above.

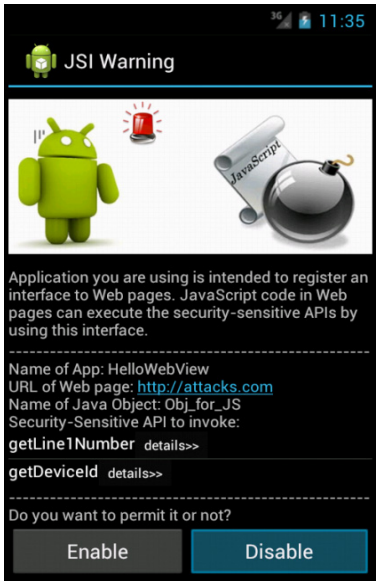


Fig. 3 Example of warning information.

Table 2 Environment of host OS.

OS	Windows 7 Home Premium
CPU	Intel(R) Core(TM) i7-3517U 1.90GHz
Memory	8 GB
Virtualization Software	VMware Player 4.0.4

Table 3 Environment of guest OS.

Distribution	Ubuntu 10.04 LTS
Kernel	Linux 2.6.32-44-generic
Number of virtual CPU	2
Memory	4 GB

5.2 Effectiveness of Modified Dexdump

We compared the unmodified and the modified dexdump. The comparison details are shown in Table 4. We used six free Android apps that downloaded from the "Recommended Apps This Week" on Google Play on June 11, 2013. The size of the assembly code generated by the modified dexdump was significantly reduced by about 90%, compared to the one generated by the unmodified dexdump.

5.3 Overhead of the Static Analysis Unit

We measured the processing time of the static analysis. The environment we used are shown in Table 2 and Table 3. We ran the guest OS using VMware Player 4.0.4 on the host machine and then ran Android 4.0.3, which the proposed method has been implemented, on the guest virtual machine. We tested two representative Android apps using WebView: HelloWebView, which is the smallest Android app that uses WebView, and LivingSocial, which has been introduced in [5] is the size of a typical Android app that uses WebView.

Table 4 Comparison between unmodified dexdump and modified dexdump.

Android App	A	B	C	D
The Weather Channel	4,944 KB	74,630 KB	5,501 KB	92.63%
Vyclone-Film together	4,659 KB	71,885 KB	6,917 KB	90.38%
Contacts+	3,131 KB	53,704 KB	5,222 KB	90.28%
Instructables	2,628 KB	41,272 KB	4,014 KB	90.27%
Sports Republic	3,531 KB	48,396 KB	3,348 KB	93.08%
Appy Gamer	2,496 KB	35,776 KB	3,081 KB	91.39%

Note: Each column shows the size of the dex file (A), the size of the assembly code generated by the unmodified dexdump (B), the size of the assembly code generated by the modified dexdump (C), and the rate of reduction (D), respectively.

Table 5 Processing times of static analysis.

Android App	Size of Dex File	C	Average of Processing Time
HelloWebView	5.6 KB	4 KB	203 ms
LivingSocial	781.8 KB	1,050 KB	8,781 ms

Note: Column C shows the size of the assembly code generated by the modified dexdump.

Processing times of static analysis are shown in Table 5. The processing times were measured from the start of converting the dex format to assembly code by using dexdump until the end of the creation of API_Class Matching List. As shown in Table 5, the processing time of HelloWebView is 203 ms, which is very short. On the other hand, the processing time of LivingSocial is about 9 s. However, because the static analysis is performed only once at installation and the installation requires relatively long time, 9 s may not be a serious inconvenience.

6. Related Works

Nowadays, JavaScript has been widely used and many works have been done to enhance the security in web browsers. Reference [8] identified the fundamental lack of fine-grained JavaScript access control mechanisms in modern web browsers and proposed a method that enables fine-grained access control in JavaScript contexts. Reference [9] presented a client-side advice implementation called CONSCRIPT, which allows the hosting page to express fine-grained application-specific security policies at runtime.

On the other hand, the researches on Android security are also booming. Reference [10] adopted bytecode rewriting to implement fine-grained access control at the API level. Reference [11] proposed DroidTrack, a method for tracking the diffusion of personal information and preventing its leakage on Android device.

Vulnerabilities caused by the use of WebView have attracted the attention of the research community [4], [7]. Reference [5] reported that WebView is used in 86% of the top 20 most downloaded Android apps in 10 different categories. Further, two attack models were discussed. Refer-

ence [6] proposed a static analysis method, to estimate the threat while using WebView. However, the threat is evaluated at the Android app level, and the user is only notified whether the Android app is dangerous or not. Therefore, the user can do nothing, except to keep the dangerous Android app unused. On the other hand, our proposed method can use Android apps with disabling suspicious Java object. As results, Android apps can run safely without calling security-sensitive APIs via JavaScript. Thus, users do not need to give up to use the Android apps.

7. Conclusions

In this paper, we described the attacks to Android devices from web pages caused by exploiting the vulnerabilities of WebView. To resolve these attacks, we proposed an access control on the security-sensitive APIs at the Java object level. The threat detection is performed when the addJavascriptInterface API is invoked to register the Java object into WebView, and the user is notified if a threat is detected. By disabling the malicious Java object, attacks from web pages could be prevented.

As a future work, we will evaluate the proposed method by using various Android apps and study a method of detecting actual threats from potential threats.

References

[1] J. Yu and T. Yamauchi, "Access control to prevent attacks exploiting vulnerabilities of WebView in Android OS," Proc. 2013 IEEE International Conference on High Performance Computing and Communications (HPCC-2013) and 2013 IEEE International Conference on Embedded and Ubiquitous Computing (EUC-2013), pp.1628–1633, 2013.

[2] Android, the world's most popular mobile platform - Android Developers. [Online]. Available: http://developer.android.com/about/index.html

[3] WebView - Android Developers. [Online]. Available: http://developer.android.com/reference/android/webkit/WebView.html

[4] Dangers lurking in the implementation of the browser features to smartphone app - issue of WebView class. (in Japanese). [Online]. Available: http://codezine.jp/article/detail/6618

[5] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin, "Attacks on webview in the android system," Proc. 27th Annual Computer Security Applications Conference (ACSAC'11), pp.343–352, 2011.

[6] H. Kawabata, T. Isohara, K. Takemori, and A. Kubota, "Threat of script abuse android permissions and static analysis," IPSJ SIG Technical Report (in Japanese), vol.2011-CSEC-53, no.3, pp.1–6, 2011.

[7] lexanderA - WebView examples. [Online]. Available: http://lexandera.com/category/webview_examples/

[8] K. Patil, X. Dong, X. Li, Z. Liang, and X. Jiang, "Towards fine-grained access control in javascript contexts," Proc. 31st International Conference on Distributed Computing Systems (ICDCS'11), pp.720–729, 2011.

[9] L. Meyerovich and B. Livshits, "CONSCRIPT: specifying and enforcing fine-grained security policies for javascript in the browser," IEEE Symposium on Security and Privacy (SP'10), pp.481–496, 2010.

[10] H. Hao, V. Singh, and W. Du, "On the effectiveness of api-level access control using bytecode rewriting in android," Proc. 8th ACM Symposium on Information, Computer and Communications Security (AsiaCCS'13), pp.25–36, 2013.

[11] S. Sakamoto, K. Okuda, R. Nakatsuka, and T. Yamauchi, "DroidTrack: Tracking and visualizing information diffusion for preventing information leakage on Android," J. Internet Services and Information Security (JISIS), vol.4, no.2, pp.55–69, 2013.