COIN-VASE: Code Injection Vulnerability Scanning Environment for HTML5-Based Android Apps

Su Yeon Choi
Seoul Women's Univ.
Seoul 01797
Republic of Korea

Jee Ah Lee
Seoul Women's Univ.
Seoul 01797
Republic of Korea

Wonhee Lee
Seoul Women's Univ.
Seoul 01797
Republic of Korea

Hae Young Lee
Seoul Women's Univ.
Seoul 01797
Republic of Korea
haelee@swu.ac.kr

ABSTRACT

Although using HTML5-based techniques to develop mobile apps provides a good solution to overcome limitations arising from multiplatform development, mobile apps developed based on the technologies are subject to code injection attacks in which malicious JavaScript code can be injected through multiple channels and then executed. This work-in-progress paper presents an environment for scanning potential code injection vulnerabilities in HTML5-based Android apps. The proposed environment performs a black-box test that injects traceable HTML tags into an app running on an emulator through internal, external, and UI channels, and then observes if some of the injected HTML tags have been triggered. The proposed environment could identify potential code injection vulnerabilities in apps, regardless of development frameworks, before they are exploited. A prototype is being developed based on our proof-of-concept.

CCS Concepts

• Security and privacy →Mobile and wireless security and web application security

Keywords

Mobile security; HTML5-based mobile apps; JavaScript; code injection attacks; vulnerability scanners.

1. INTRODUCTION

Using HTML5-based techniques to develop mobile apps provides a good solution to overcome limitations arising from multiplatform development (e.g., Android and iOS), so that HTML5-based mobile apps are becoming more and more popular since [1]. These apps are implemented using HTML5 and JavaScript [2], where code and data can be mixed together since the code can be automatically identified and then executed by JavaScript engines. Like this feature has led to Cross-Site Scripting (XSS) attacks on web (HTTP) applications, HTML5-based apps are also subject to the similar code injection attacks through 'many' channels, including Bluetooth, Contacts, SMS, Wi-Fi, and so on [2]. An example of such code injection attacks can be found at [3].

In order to automatically scan HTML5-based Android apps that are vulnerable to code injection attacks, Jin *et al*. [1] proposed a solution for apps developed using Adobe PhoneGap [4], in which static taint analysis is used to find code injection vulnerabilities within the app's JavaScript code that is rewritten and then sliced. However, their solution can scan only PhoneGap-based Android apps whose source code is available. Mao *et al*. proposed a solution [5] to detect code injection attacks in HTML5-based Android apps by monitoring the execution of the apps and generating behavior state machines to describe the apps' runtime behaviors based on the execution contexts of the apps. However, their solution cannot detect code injection vulnerabilities unless they are exploited.

In this paper, we propose a COde INjection VulnerAbility Scanning Environment (COIN-VASE) for HTML5-based Android apps, which is based on our initial work [6]. For an app running on an emulator, HTML `script` tags for checking potential code injection vulnerabilities are injected into internal resources, such as the file system, metadata fields, data shared among apps (e.g., `Contacts`), and so on, through the external channels, including SSIDs of Wi-Fi access points, names of Bluetooth devices, and NFC data, and through the UI of the app (e.g., `text`, `password`, `textarea` fields). While an automated UI testing tool is exploring the UI, some of the injected HTML tags may be triggered. These triggering events are logged by an external server, so that the environment can check if the app has potential code injection vulnerabilities and which of the injected HTML tags were triggered. Unlike Jin *et al*.'s solution [1], the proposed environment can scan an app without its source code, regardless of its development framework. Differing from Mao *et al*.'s runtime solution [5], it could identify potential code injection vulnerabilities before they are exploited. We have just implemented a proof-of-concept and are building a prototype.

2. 2. COIN-VASE

2.1 Structure

Figure 1 shows the conceptual structure of COIN-VASE. (a) COIN-VASE is designed to automatically scan potential code injection vulnerabilities in an HTML5-based Android app that is running on an emulator, e.g., Genymotion [7]. (b) Before scanning, HTML `script` tags for checking the vulnerabilities are first injected into internal resources by an app of COIN-VASE. The injected `script` tags specify JavaScript (JS) files on a web server. In order to enable the server to identify which of the injected tags were triggered, each of the tags (hereafter called 'markers') specifies a unique URL on the server. (c) COIN-VASE also sets the emulator inject markers through the external channels while scanning. (d) The UI of the app is then explored by an automated UI testing tool. While exploring, the tool also tries to inject markers through the UI, e.g., `text`, `password`, `textarea` fields. During the exploration, some of the markers could be displayed through `WebView` without sanitation if the

MobiCASE 2016, November 30-December 01, Cambridge, Great Britain

Copyright © 2016 EAI

DOI 10.4108/eai.30-11-2016.2267038

app has the vulnerabilities. (e) Then, they will be triggered, which will lead to sending HTTP requests for JS files to the server. These requests are logged by the server, so that the user can check if the vulnerabilities exist in the app, and which of the markers were triggered.

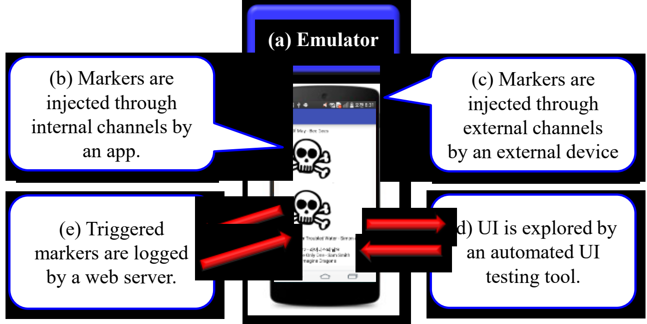

Figure 1. Conceptual structure of COIN-VASE.

Please use a 9-point Times Roman font, or other Roman font with serifs, as close as possible in appearance to Times Roman in which these guidelines have been set. The goal is to have a 9-point text, as you see here. Please use sans-serif or non-proportional fonts only for special purposes, such as distinguishing source code text. If Times Roman is not available, try the font named Computer Modern Roman. On a Macintosh, use the font named Times. Right margins should be justified, not ragged.

2.2 Markers

A marker is a script tag ‘`<script src="http://a.io/xx.js">`,’ where *xx* is a unique number for an injection channel. For channels that do not accept HTML tags (e.g., the file system), markers are injected through URL encoding. For example, in order to inject a marker into the file system, ‘<’, ‘"’, ‘:’, ‘/’, and ‘>’ characters in the marker can be replaced with ‘`%3C`’, ‘`%22`’, ‘`%3A`’, ‘`%2F`’, and ‘`%3E`’ characters, respectively. Also, before scanning, COIN-VASE edits hots files (e.g., *aaa.bbb.ccc.ddd* `a`) in order to shorten markers; for example, the marker may be ‘`%3Cscript src=%22http%3A%2F%2Fa%2F2.js%22%3E.mp3`.’

2.3 Marker Injections

An app of COIN-VASE injects markers through the internal channels: 1) Content Provider, e.g., `Calendar`, `Call logs`, `Contacts`, `Messaging`, `Notes`, and so on. For example, the app creates a calendar event in which markers are placed in the title and details. 2) Metadata, e.g., title, artist, and album. For example, the app places some multimedia files, such as JPEG, MP3, and MP4 files, in which markers are placed in the metadata fields. 3) File System. The app creates some files, including directories, of which names are markers, or that contain markers.

Markers can be injected by an external device through the external channels: 1) an SSID that is a marker, 2) a Bluetooth device's name that is a marker, and 3) an NFC marker by host card emulation. COIN-VASE provides an external device that injects such markers. Meanwhile, markers can be also injected through the UI of the app while the UI is explored by an automated UI testing tool.

2.4 Triggered Marker Scans

In order to scan potential code injection vulnerabilities, the UI of the app is explored by an automated UI testing tool. While the UI is being explored, some of the injected markers could be displayed through `WebView` without sanitation, i.e., potential code injection vulnerabilities exist in the app. The markers then will be automatically identified as code, and thus sent to the JavaScript engine for execution, which will lead to sending HTTP requests for JS files associated with the triggered markers to the server. These triggered markers are logged by the server, so that the users can review potential code injection vulnerabilities found in the app through a web interface of the server.

3. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed an environment that scans potential code injection vulnerabilities in HTML5-based Android apps, by injecting markers through internal, external, and UI channels and checking triggered markers. Compared to the existing solutions to code injection attacks, the proposed environment can scan any Android app for potential code injection vulnerabilities without its source code before they (if any) are exploited. A prototype is being built based on our proof-of-concept.

The proposed environment, however, cannot identify potential code injection vulnerabilities from some channels, e.g., from communication channels (e.g., mobile data, Wi-Fi, Bluetooth, and NFC) or sensors (e.g., cameras), and so on. Once a prototype has been built, we will investigate solutions to these limitations.

4. ACKNOWLEDGEMENTS

This research was supported by the MISP (Ministry of Science, ICT & Future Planning), Korea, under the National Program for Excellence in SW (R7116-16-1018) supervised by the IITP (Institute for Information & communications Technology Promotion). (R7116-16-1018)

5. REFERENCES

[1] Jin, X., Hu, X., Ying, K., Du, W., and Yin, H. 2014. Code Injection Attacks on HTML5-based Mobile Apps: Characterization, Detection and Mitigation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, AZ, USA, November 03 -07, 2014). CCS '14. ACM, 66-77. DOI=http://dx.doi.org/10.1145/2660267.2660275.

[2] Georgiev, M., Jana, S., and Shmatikov, V. 2015. Rethinking Security of Web-Based System Applications. In *Proceedings of the 24th International Conference on World Wide Web* (Florence, Italy, May 18-22, 2015). WWW '15. ACM, 366-376. DOI=http://dx.doi.org/10.1145/2736277.2741663.

[3] Code Injection Attacks on HTML5-based Mobile Apps. http://www.cis.syr.edu/~wedu/android/JSCodeInjection/.

[4] Adobe PhoneGap. http://phonegap.com/.

[5] Mao, J., Wang, R., Chen, Y., and Jia, Y. 2016. Detecting Injected Behaviors in HTML5-Based Android Applications. *Journal of High Speed Networks* 22, 1 (February 2016), 15-34. DOI=http://dx.doi.org/10.3233/JHS-160534.

[6] Choi, S.Y., and Lee, H.Y., Toward Automated Scanning for Code Injection Vulnerabilities in HTML5-Based Mobile Apps. In *Proceedings of the 2nd International Conference on Software Security and Assurance* (St. Pölten, Austria, August 24-25, 2016). ICSSA '15. IEEE, to be published.

[7] Genymotion. https://www.genymotion.com/.