Windows Phone 8.1: WebView Security

Bogdan Copos
Computer Science Department
University of California, Davis
Email: bcopos@ucdavis.edu

Minghua Zhu
Computer Science Department
University of California, Davis
Email: mhzhu@ucdavis.edu

***Abstract*—Smartphone platforms have become an intriguing target for malicious attackers, specifically due to the amount of sensitive information available on devices. While the Android platform has been extensively studied, Windows Phone platform has received little attention from researchers. In this paper, we present a framework designed to analyze the security of Windows Phone applications which use a WebView component and implement Javascript handlers, a known attack vector in the Android ecosystem. Our results show that few applications use the WebView component but out of those 80% applications use Javascript handlers. Furthermore, [something about calling uri]**

I. Introduction

Smartphone platforms have become an intriguing target for malicious attackers, specifically due to the amount of sensitive information available on devices. Due to its initial lack in popularity among smartphone users, the Windows Phone platform has received little attention from researchers and security specialists. However, the platform has similarities to other smartphone platforms such as Android and may be vulnerable to similar attacks. One of the most prevalent vulnerabilities in the smartphone environment is data leakage. Data leakage attacks can happen in several ways. One way involves malicious application developers who construct an application with the appropriate capabilities so that it may access users sensitive data and send it back to the attackers. Another popular attack vector is through the WebView component. The WebView is a component which allows developers to display web content such as HTML pages and Javascript. Often, to extend the usability of such web-based applications, the developer may want to access system resources through Javascript. This is allowed but may create vulnerabilities in the application. For example, if the WebView has Javascript enabled and allows access to system resources, any Javascript (whether written by the developer or simply a third-party) on a website accessed by that WebView, also has access to that sensitive information. In this paper, we present an analysis tool which aims at detecting such vulnerabilities in Windows Phone 8.1 applications. In addition, we describe a Windows Phone store crawler which allowed us to gather 490 applications. The collected applications were used to test our analysis tool.

II. Background

Previous works which have studied security of data in smartphone platforms analyze applications can split into two categories. One category assumes a careless or perhaps even malicious application developer. In such cases, sensitive data extracted by the application using permissions to device resources is leaked off of the device via SMS, MMS or any other method for transmitting data externally. The work of William Enck et. al. presented in TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones falls under this category. As part of their work, they developed a taint-tracking system for Android applications. The taint-tracking system works by tracing data through the application during execution. Specifically, the applications sources and sinks are specified. A source in this case is any API function call which results in sensitive data being obtained. Some examples of sensitive data are device owners contacts and media files. Sinks are also API functions which transmit data off of the device. Some examples of sinks are the sendSMS() and sendMMS() functions. Once sources are defined, the application is executed and exercised. During the execution, the taint tracking system follows data from sources to see if they ever enter any of the defined sinks. TaintDroid, as a dynamic approach, involves a modified version of the Androids Dalvik Virtual Machine which allows for the tagging of data during execution. This technique allows TaintDroid to track data across applications (i.e. inter-process communication) as well as when an application calls native code (e.g. third party library shared object). While this method adds some overhead, the results are more accurate than that of static taint tracking or data flow analysis tools. Furthermore, TaintDroid was then tested on 30 random, popular, third-party applications. The results showed that two thirds of the applications studied display suspicious handling of sensitive data. Moreover, the findings show that half of the applications studied released sensitive data such as GPS location to third-party advertising companies through advertisement libraries.

The second category assumes a benign application which is attacked by third party code, either through Javascript or inter-process communication. Example of works in this category are Luo et. al.s Attacks on WebView in the Android and Bifocals: Analyzing WebView Vulnerabilities in Android Applications where Erika Chin et. al. presented a tool for examining Android applications and their WebView component. Specifically, the Bifocals tool detected vulnerabilities to excess authorization attacks and file-based cross-zone scripting attacks. Excess authorization attacks are attacks where malicious Javascript can invoke application code (via Javascript interfaces). When a WebView enables JavaScript, registers a JavaScript interface, JavaScript content in the WebView can invoke the registered interfaces. If malicious third-party JavaScript by any chance (through a man-in-the-middle attacker or Malicious third-party content) gets loaded in the page, then it can also invoke the applications interface. Depending on the functionality of the application code, excess authorization vulnerability can cause different security implications such as Information injection

and leakage. File-based cross-zone scripting attacks occur when Javascripts have access to the devices file system. The Android WebView renderer treats everything loaded via a file:// URL as being in the same origin. If an application loads static content via a file:// URL, and unfortunately this content includes malicious JavaScript, this JavaScript has the ability to access all the files in the file system that the application can. Bifocals can automatically recognize WebView vulnerabilities in Android applications by analyzing two aspects of WebView interaction, the application and the web content. Bifocals has found 67 out of 608 (11%) android applications with WebView-related vulnerabilities.

While similar in scope to the works listed above, our proposed project differs in that it applies to the Windows Phone 8.1 platform. It is not guaranteed that vulnerabilities in Android also exist on the Windows platform. For example, one main difference between the two platforms involves their permission (capabilities in Windows) mechanism. With respect to Javascript attacks, the Windows Phone 8.1 platform does have a similar WebView component. However, the presence of the WebVIew component does not imply the underlying implementation and security mechanisms are identical. From our literature search, no previous work has investigated the above listed attack vectors on the Windows Phone 8.1 platform.

The Windows Phone platform allows applications to be written in a number of languages, including C#, Visual Basic, and even HTML and Javascript. Before being published to the application store, the phones are packaged. The format of the package depends on the target version. Applications targeting Windows Phone 8 and above, will use the new package format, *appx*. This format resembles the *zip* format and unlike the old application package format *xap*, the application packages are not encrypted and can easily be unpackaged. An application package will include the application binary, in MSIL/CIL format, any supporting libraries, necessary media files, as well as manifest files describing the application and its properties such as the required capabilities, prerequisites, resources and even visual elements (similar to activities in the Android platform).

III. Methodology

Our analysis consisted of two main components. The first component is an application crawler. The second component is responsible for filtering the applications gathered and performing the analysis.

A. Application Crawler

Since our analysis focuses on a small subset of applications (i.e. with WebView and JS handler), we needed to gather as many applications as possible. The Windows Phone store application crawler enables us to achieve this goal. The crawler takes advantage of the fact that Microsoft's Bing search engine indexes the Windows Phone application store. This allows a Bing user to search the application store. Furthermore, using a special query, any user can obtain download links for all Windows Phone store applications of the newer *appx* format. The *appx* format started being used in version [VERSION] of the Windows Phone platform. Our crawler works by accessing the Bing search engine, providing the special query and crawling the Bing result pages for the download links. Once the links are gathered, the downloading of application packages begins. The contents of the application packages include the application binary (in MSIL/CIL format) and other resources and they can easily be extracted using a zip archive utility.

B. Analysis

IV. Results

Unfortunately, despite gathering 490 applications, the number of applications which met our prerequisites for the applications is only 4. Out of the 490 applications, only 5 applications use a WebView component. 4 out of those 5 applications also implement a Javascript handler. 3 of the 4 applications which have a Javascript handler are different versions of the same application. Our analysis also determined that none of the Javascript handlers discovered called a sensitive API function, directly or indirectly.

Furthermore, out of the 4 applications, [NUMBER] applications check the "calling URI" within their Javascript handlers. Although it could be more dangerous and detrimental if the handlers executed a sensitive API, not verifying the "calling URI" is considered bad practice.

V. Challenges

VI. Conclusion

The conclusion goes here.

Acknowledgment

The authors would like to thank our sponsor, Yekaterina O'Neil, at Hewlett Packard, the organizers of the INSURE program and our mentor Professor Matt Bishop.