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 the devices. While the Android platform has been extensively studied, the 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 the WebView component and implement Javascript handlers. The bridge between Javascript and applications is a known attack vector in the Android ecosystem. Our results show that few applications use the WebView component. Out of the applications which include at least a WebView component, 80% also use Javascript handlers. Our analysis detected a few cases in which the Javascript can cause the application to display the phone call user interface (UI). Perhaps more alarmingly, the Javascript can control the number automatically inserted into the UI. Furthermore, our analysis discovered that none of the applications using Javascript handlers verify the calling URI, allowing any third party content to access the same handlers.**

I. Introduction

Smartphone platforms have become an intriguing target for malicious attackers, specifically due to the amount of sensitive information available on the 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 vulnerability classes 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 capabilities necessary to access users sensitive data. Once collected, the data is sent back to the attackers. Another popular attack vector is through the WebView component. The WebView is a component which allows developers to display remote 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 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 390 applications. The collected applications were used to test our analysis tool.

II. Background

Previous works which have studied the security of data in smartphone platforms can be split into two categories. One category assumes a careless or perhaps even malicious application developer. In such cases, sensitive data is extracted by the application using permissions and is transmitted off 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 [3] falls under this category. As part of their work, they developed a dynamic 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 gathered. Some examples of sensitive data are contacts and media files. Sinks are 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 the sources to determine if it ever enters a 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 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 [4] and Bifocals: Analyzing WebView Vulnerabilities in Android Applications [2] where Erika Chin et. al. presented a tool for examining Android applications and their WebView components. Specifically, the Bifocals tool detected vulnerabilities which make excess authorization attacks and file-based cross-zone scripting attacks possible. Excess authorization attacks are attacks where malicious Javascript invokes application code (via Javascript interfaces). When a WebView enables Javascript and registers a Javascript interface, Javascript content in the

WebView can invoke the registered interfaces. Any malicious third-party Javascript which gets loaded in the WebView page (for example through a man-in-the-middle attack) can also invoke the applications interface. Depending on the functionality of the application code, excess authorization vulnerability can cause various 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. 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 or that the same vulnerabilities exist in the Windows Phone platform. 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 use the new package format, *appx*. This format resembles the *zip* format. Unlike the old application package format *xap*, the *appx* 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 and necessary media files. An application package will also contain 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). As mentioned above, applications may contain a WebView component. Such components are used to display remote web content. To extend the functionality of the WebView components, Javascripts may be executed and can interact with the application code via Javascript handlers. An example of a Javascript handler can be seen in Listing 1. There we see how a Javascript handler is associated with a specific WebView object via the *addEventListener()* function call. Below that we see the actual Javascript handler and the contents of its body. Notice how the developer of the application can choose to check the calling URI inside the Javascript handler. Such practice enforces the Same Origin Policy.

```
1 webview.addEventListener("MSWebViewScriptNotify",
      handleScriptNotifyEvents);
2 function handleScriptNotifyEvents(e) {
3         if (e.callingUri === "https://msgnotify.
      example.net/") {
4             if(e.value === "msg1")
5             {
6                 // Process the message.);
7             }
8         }
9     }
```

Listing 1. An example of a Javascript handler

III. Methodology

Our analysis consists 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), it is crucial that we 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 query the application store. More specifically, using a special query, any user can obtain download links for all Windows Phone store applications of the newer *appx* format. 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. As mentioned earlier, the contents of the application packages include the application binary (in MSIL/CIL format) and other resources. An application package can easily be extracted using a zip archive utility.

B. Analysis

The analysis is done directly on the application binaries using the Cecil library [1]. Cecil is a library used to generate and inspect binaries in the MSIL format. The library is built on top of Mono, an open source .NET framework alternative [5] Using Cecil, we sequentially read and analyze the assembly of every application. Listing 2 shows how Cecil is used to parse a MSIL binary. The *AssemblyDefinition.ReadAssembly* function is used to read the binary. Lines 7-13 show how the assembly is then queried for all Methods and all instructions within those methods.

```
1 try:
2   asm = AssemblyDefinition.ReadAssembly("\some\app.
      exe")
3 except SystemError:
4   print "FAILED: " + str(os.path.join(f_dir, f))
5   return
6
7 for module in asm.Modules:
8   for type in module.Types:
9     for method in type.Methods:
10      print method.FullName
11      if method.Body:
12        for ins in method.Body.Instructions:
13          print ins
```

Listing 2. Example of Cecil in Python

First, the assembly is parsed for prerequisites. Specifically, we check that the applications uses the WebView component. If so,

our analysis verifies that the application implements Javascript handlers. Last but not least, it also searches for sensitive API functions among the methods called by these handlers. If none are found, the analysis process ends. If sensitive functions are called, the analysis performs intra-procedural data flow analysis. The objective here is to determine if data from the Javascript (passed via arguments to the Javascript handlers) flows into arguments of the sensitive API functions. Intra-procedural refers to the fact that the data flow analysis is performed individually for every method, compared to inter-procedural analysis, where methods are inlined beforehand. The data flow analysis is done by stepping through the instructions of a given method and simulating the stack. For this, our analysis has the behavior of every MSIL opcode encoded such that when a specific instruction is reached, the appropriate steps are taken with respect to the stack. To detect if data flows from the Javascript to a method call, our analysis looks to see if inside the Javascript handler, the arguments are loaded (using the *ldarg* instruction) and then later used by a *call* instruction of a sensitive API function. Since this may not always be a direct data transfer, our analysis mimics the process of taint analysis. For example, let's say an object is created inside the Javascript handler. One of the parameters (or its value) of the Javascript handler is then stored as a field of the newly created object. Later in the handler's body, the object is passed to a sensitive API as an argument. Our analysis is capable of detecting such cases by marking the object (the object reference, more specifically) as tainted, as soon as the field of the object represents the value of the handler's parameter.

IV. Results

Unfortunately, despite gathering 390 applications, only 4 applications met our prerequisites. Out of the 390 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 these 3 applications call the *ShowPhoneCallUI* method. Also, the number inserted into the phone UI is controlled by the Javascript.

None of the 4 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

Applications : Windows phone applications can be written in a number of languages such as C#, Visual Basic, HTML and Javascript—which makes our analysis difficult. Also, since applications can be written directly in HTML and Javascript, such applications will not have a WebView component, limiting the sample set we can test our framework in.

Analysis tools : Windows Phone applications are written in C#, compiled to an intermediate language and then packaged into the ".appx" format. While the unpacking is easy, interpreting the intermediate language (MSIL/CIL) files is not easy. To perform the analysis, we must learn what every instructions opcode does and how it affects the stack. This process is tedious and a small mistake can skew our results.

Time : This course is only 16 weeks long. Because of such short time table, any unexpected difficulties can greatly impact the deliverables of this project.

VI. Future Work

While our analysis tools focuses on Javascript handlers, we believe the data flow analysis can also be applied to investigate over-privileged applications which leak sensitive information. In the future, we would like to extend our framework to also search for such applications.

Additionally, we would like to change our data flow analysis from intra-procedural to inter-procedural for more accurate analysis.

VII. Conclusion

In this paper, we present we present a framework designed to analyze the security of Windows Phone applications which use the WebView component and implement Javascript handlers. Our tool is capable of detecting a few cases in which the Javascript can cause the application to display the phone call user interface (UI). Perhaps more alarmingly, the Javascript can control the number automatically inserted into the UI. We tested our tool on 390 applications. The results show that five applications use the WebView component. Out of the applications which include at least one WebView component, 80% also use Javascript handlers.

Acknowledgment

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

References

[1] Cecil. http://www.mono-project.com/docs/tools\+libraries/libraries/Mono.Cecil.

[2] E. Chin and D. Wagner. Bifocals: Analyzing webview vulnerabilities in android applications. In *Information Security Applications*, pages 138–159. Springer, 2014.

[3] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[4] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin. Attacks on webview in the android system. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 343–352. ACM, 2011.

[5] Mono: Cross-platform, open source .net framework. http://www.mono-project.com/.

Appendix

Here is the list of API functions we consider sensitive for the purpose of this project. Essentially, the list is composed of any function which either accesses user data (whether it's contacts or files from storage or SD card), accesses system resources (e.g. sensors, camera, etc.) or communicates with the outside world (via SMS, email, Bluetooth, etc.).

```
windows.devices.geolocation.geolocator.
    getgeopositionasync
windows.devices.geolocation.geolocator.
    getgeopositionasync(timespan,timespan)
```

```
windows.devices.geolocation.geolocator.getgeopositionhistoryasync(datetime)
windows.devices.geolocation.geolocator.getgeopositionhistoryasync(datetime,timespan)
windows.devices.geolocation.geolocator.requestaccessasync

windows.devices.radios.radio.requestaccessasync

windows.applicationmodel.calls.phonecallmanager.showphonecallui

windows.applicationmodel.chat.chatmessagemanager.showcomposesmsmessageasync
windows.applicationmodel.email.emailmanager.showcomposenewemailasync

windows.applicationmodel.contacts.contactstore.getcontactasync
windows.applicationmodel.contacts.contactstore.getmecontactasync
windows.applicationmodel.contacts.contactstore.getcontactlistasync
windows.applicationmodel.contacts.contactstore.findcontactsasync
windows.applicationmodel.contacts.contactreader.getmatchingpropertieswithmatchreason
windows.applicationmodel.contacts.contactreader.readbatchasync
windows.applicationmodel.contacts.contactlist.getcontactasync
windows.applicationmodel.contacts.contactlist.getmecontactasync

windows.devices.wifi.wifiadapter.connectasync
windows.devices.wifi.wifiadapter.requestaccessasync
windows.devices.wifi.wifiadapter.scanasync

windows.storage.applicationdata.current.localsettings
windows.storage.applicationdata.current.localfolder.getfileasync
windows.storage.applicationdata.current.localfolder.createfileasync
windows.storage.applicationdata.clearasync
windows.storage.applicationdata.getforuserasync
windows.storage.downloadsfolder.createfileasync
windows.storage.downloadsfolder.createfolderasync

windows.devices.sensors.accelerometer.getcurrentreading
windows.devices.sensors.activitysensor.getcurrentreadingasync
windows.devices.sensors.altimeter.getcurrentreading
windows.devices.sensors.barometer.getcurrentreading
windows.devices.sensors.compass.getcurrentreading
windows.devices.sensors.gyrometer.getcurrentreading
windows.devices.sensors.inclinometer.getcurrentreading
windows.devices.sensors.lightsensor.getcurrentreading
windows.devices.sensors.magnetometer.getcurrentreading
windows.devices.sensors.pedometer.getsystemhistoryasync
windows.devices.sensors.proximitysensor.getcurrentreading

windows.media.capture.cameracaptureui.capturefileasync
windows.media.capture.mediacapture.capturephototostoragefileasync
windows.media.capture.mediacapture.capturephototostreamasync
windows.media.capture.mediacapture.startrecordtostoragefileasync
windows.media.capture.mediacapture.startrecordtostreamasync
windows.media.capture.screencapture.getforcurrentview

windows.media.speechrecognition.speechcontinuousrecognitionsession.startasync

windows.networking.backgroundtransfer.backgrounddownloader.createdownload
windows.networking.backgroundtransfer.backgrounddownloader.createdownloadasync
windows.networking.backgroundtransfer.backgrounduploader.createupload
windows.networking.backgroundtransfer.backgrounduploader.createuploadasync
```