Taming Web Views in the Detection of Android Privacy Leaks

Xue Qin
xue.qin@utsa.edu
The University of Texas at San Antonio
San Antonio, Texas

Robert Neuhaus
robert.neuhaus@my.utsa.edu
The University of Texas at San Antonio
San Antonio, Texas

Diego Gonzales
diego.gonzales@my.utsa.edu
The University of Texas at San Antonio
San Antonio, Texas

Xiaoyin Wang
xiaoyin.wang@utsa.edu
The University of Texas at San Antonio
San Antonio, Texas

Travis Breaux
breaux@cs.cmu.edu
Carnegie Mellon University
Pittsburgh, Pennsylvania

Jianwei Niu
jianwei.niu@utsa.edu
The University of Texas at San Antonio
San Antonio, Texas

ABSTRACT

Billions of smartphone users force both technical and non-technical facilities to publish their applications in market. One of the easiest ways to create an application for non-techies is by transferring their existing website using WebView. Current approaches for privacy policy analysis are mainly focused on APIs and predefined user interfaces and cannot cover these type of information. In this paper, we proposed a novel solution to trace the input data that is collected from WebView.

CCS CONCEPTS

• **Security and privacy → software and application security**.

KEYWORDS

android applications, privacy policies, violation detection

ACM Reference Format:
Xue Qin, Robert Neuhaus, Diego Gonzales, Xiaoyin Wang, Travis Breaux, and Jianwei Niu. 2019. Taming Web Views in the Detection of Android Privacy Leaks. In *Hot Topics in the Science of Security Symposium (HotSoS), April 1–3, 2019, Nashville, TN, USA*. ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3314058.3317732

1 INTRODUCTION

Existing work by Slavin et al. [2] attempts to trace privacy policy statements about collection of platform information to API calls with static program analysis. These API calls concern personal data that is automatically collected from the device, such as user location, device identifiers, contact information, and sensor data. Prior work by Wang [3] focuses on tracking privacy policy claims based on direct user input. The user interfaces related to these inputs are generated either statically or programmatically at runtime. However, these works are limited because they did not address personal data that collected through WebView generated user interfaces, which is another major source of private information.

HotSoS, April 1–3, 2019, Nashville, TN, USA

ACM ISBN 978-1-4503-7147-6/19/04…$15.00
https://doi.org/10.1145/3314058.3317732

Traditional graphic user interface (GUI) can be implemented using static declarations in resource files, or programmatically in the code. But they are all generated locally. WebView displays web pages as part of the GUIs by loading URLs and it also supports the interaction in the opposite direction, from JavaScript to Java. In other words, the GUIs that is generated remotely by server makes static analysis impossible.

Nowadays, many technical teams provide services for non-technical organizations such as universities and hospitals to take their existing web applications and convert them into mobile apps with the help of WebView. Company like GoNative.io[1] helped their clients publish 600+ apps with a 98.1% approval rating in 2018. To address the problem, we propose a novel approach to trigger the application loading WebView at runtime while tracing the information flows among the WebView interfaces.

2 APPROACH

Our approach contains three major parts. The first part is to develop a tool to automatically explore the Android application while supporting both traditional and WebView GUI analysis. The second part is the procedure of WebView interface detection, we need to identify and modify all the interfaces statically so that they can be recognized by our GUI explorer at runtime and keep track of the information types. The last part is about data flow monitoring, we apply all the potential interface methods to Flowdroid [1] to trace information flow.

2.1 GUI Exploration

In order to automatically explore the GUIs of an Android application, we develop a GUI automation tool based on a test automation framework Appium with combination of Android Debug Bridge (adb) and Android UIAutomator. The fundamental goal of our tool is to replace the manual exploration as well as to support WebView GUI analysis.

The tool follows a series of steps to explore: First, our tool takes application as input and preprocess the setups on Appium. For the app that requires login, we provide a manual script that contains login information such as username and password; Then, our tool tries to trigger every possible GUI event on the screen by clicking and entering input data. Subsequently, our tool keeps track of all

[1] https://gonative.io/about

the GUI events it has been explored and marks down all the visited WebView within a timeout threshold. The final step is to report the visited record.

2.2 WebView Interface Detection

WebView provides a mechanism for the JavaScript code inside it to invoke Android apps' Java code. The API used for this purpose is called *addJavascriptInterface()*. Android applications can register Java objects to WebView through this API, and all the public methods in these Java objects can be invoked by the JavaScript code from inside WebView. Since Android API 17, all Java methods must include the *@JavascriptInterface* annotation to be accessible within JavaScript.

```
1  public class MainActivity extends AppCompatActivity {
2      @Override
3      protected void onCreate(Bundle savedInstanceState) {
4          super.onCreate(savedInstanceState);
5          setContentView(R.layout.activity_main);
6          WebView myWebView = (WebView) findViewById(R.id.webview);
7          myWebView.addJavascriptInterface(new WebAppInterface(this), "Android
                   ");
8          WebSettings webSettings = myWebView.getSettings();
9          webSettings.setJavaScriptEnabled(true);
10         myWebView.loadUrl("file:///android_asset/js.html");
11     }
12
13     public class WebAppInterface {
14         Context mContext;
15         WebAppInterface(Context c) {
16             mContext = c;
17         }
18         @JavascriptInterface
19         public void showToast(String toast) {
20             Toast.makeText(mContext, toast, Toast.LENGTH_SHORT).show();
21         }
22     }
23 }
```

Listing 1: Java Code of WebView Interface Implementation

Listing 1 shows a GUI implementation example using WebView. keyword *@JavascriptInterface* in line 18 annotates the *showToast()* method of the *WebAppInterface* class, and Line 7 illustrates the use of the *addJavascriptInterface()* API method with an instance of the *WebAppInterface* class included as a parameter. The *addJavascriptInterface()* method allows the *WebAppInterface* instance to be accessible from within JavaScript, while the *@JavascriptInterface* annotation allows JavaScript to call the instance's *showToast()* method. Line 6 in Listing 2 illustrates the *showToast()* method of the *WebAppInterface* instance being called from within JavaScript, passing in a string as a parameter. Without the *@JavascriptInterface* annotation, the *showToast()* method would not be callable from within JavaScript.

The fact that any method that is called within JavaScript must contain a specific annotation means that a list of methods to monitor can be generated by decompiling an Android application and parsing the resulting .smali files for methods containing the annotation. Figure 1 shows us the frequency of WebView annotation usage from top 100 apps across 8 categories. From the result, 74% of the apps are using this annotation. And 20 applications each consisted of at least 25 methods containing the annotation, and 5 applications have more than 100 annotations.

```
1  <head>
2      <title>Webview Test</title>
3  </head>
4  <script type="text/javascript">
5      function showAndroidToast(toast) {
6          Android.showToast(toast);
7      }
8  </script>
9  <body>
10 <input type="button" value="Say␣hello" onClick="showAndroidToast('Hello␣
       Android!')" />
11 </body>
```

Listing 2: Sample JavaScript Code of a Java Object's Method Being Called

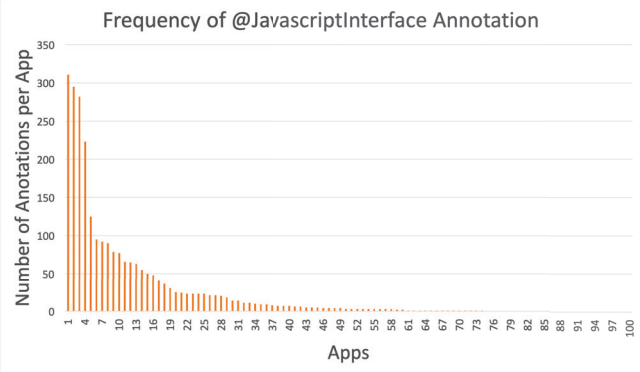

Figure 1: Frequency of *@JavascriptInterface* Annotation

2.3 Data Flow Monitoring

After we identify the all web view interface method invocations between JavaScript code and Android Java code, we will instrument the invocations and record the data transferred through the interface. During the GUI exploration, we input a unique value into each different text box, so that we can tell the source of the data based on the value. Therefore, for each interface invocation, we can tell which text boxes are its data source. Finally, applying FlowDroid to the Android Java code with the interface invocations as information sources, we can tell input of which text boxes in the web views flows to the network.

REFERENCES

[1] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. *SIGPLAN Not.* 49, 6 (Jun 2014), 259–269. https://doi.org/10.1145/2666356.2594299

[2] Rocky Slavin, Xiaoyin Wang, Mitra Bokaei Hosseini, James Hester, Ram Krishnan, Jaspreet Bhatia, Travis D. Breaux, and Jianwei Niu. 2016. Toward a Framework for Detecting Privacy Policy Violations in Android Application Code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/2884781.2884855

[3] X. Wang, X. Qin, M. Bokaei Hosseini, R. Slavin, T. D. Breaux, and J. Niu. 2018. GUILeak: Tracing Privacy Policy Claims on User Input Data for Android Applications. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 37–47.