Vulnerability Parser: A Static Vulnerability Analysis System for Android Applications

Yingxian Chang², Bin Liu¹, Lianri Cong¹, Hua Deng²*, Jiaming Li³ and Yuanzhe Chen³

¹ State Grid Shangdong Electric Power Company YanTai Power Supply Company
² State Grid Shangdong Electric Power Company
³ Department of Computer Science and Technology Harbin Institute of Technology
* E-Mail: denghua88@163.com (Hua Deng)

Abstract. In the case of user information leakage, the security problem of Android applications is of great importance. How to quickly and efficiently detect Android application security vulnerabilities has become an urgent research topic in security field. Aiming at the key technologies such as vulnerability detection in Android application, architecture is proposed for static analysis of android applications, namely the Vulnerability Parser. The architecture is composed of the following components, such as APK Decompressor, Manifest.xml Parser, Vulnerability Vector, and DexParser etc. Vulnerability Vector is extensible and detection rules can be configured for different vulnerabilities. The implementation of primary components is described in detail. Preliminary experimental results demonstrate its effectiveness.

1. Introduction

Nowadays, mobile usage is growing rapidly, it is estimated that there are more than 2 billion apps in Apple App Store and 2.2 million in Google Play Store. The latest research shows that 25% of mobile apps contains at least one high-risk security vulnerability, while 59% of finance app on Android had three OWAS Mobile top 10 risks. There are multiple types of vulnerabilities, and some of the dangers are as follows, e.g. Leaking personal user sensitive data (email, credential, IMEI, GPS, MAC address) over the app, communication over the network with little or no encryption, Having world readable/writable file, arbitrary code execution, and Malware [1].

Static analysis is usually used for vulnerability detection. In contrast to dynamic detection approach [2-3], the App’s code is statically analyzed for malicious activities or threats. Representative work includes: CHEX [4] is a tool to detect component hijacking vulnerabilities in the app by using taints tracking, SCandroid [5] a tool for generating automated security certification for the app. ComDroid [6] is used for communication-based threats, Amandroid [7] is designed for inter-component data analysis, and some other approaches etal [8-10].

In this paper, architecture is proposed for vulnerability analysis for android applications, namely the Vulnerability Parser. The implementation of its primary components are discussed and described in detail.
2. The Architecture of Vulnerability Parser

![Figure 1. The Architecture of Vulnerability_Parser.](image)

The following components are included such as DexParser, Vulnerability Vector, Manifest.xml Parser and APK Decompressor, etc. Vulnerability Vector is extensible and detection rules can be configured for different vulnerabilities, such as WebView FileAccess & ExportedComponents vulnerability, ContentProvider & Directory Traversal, Dynamically Registered Unprotected Broadcast Receiver, etc. Three engines are designed for Dex Parser, e.g. String Search Engine, Filter Engine, and Parser Engine.

In the case of user privacy leakage, the security of Android applications is of great importance. How to quickly and efficiently detect Android application vulnerabilities has become an urgent topic in security field. Aiming at the vulnerability detection in Android application, architecture is proposed by using static analysis, namely Vulnerability Parser. The architecture is depicted in Fig. 1, composing of the following components such as DexParser, Vulnerability Vector, Manifest.xml Parser, and APK Decompressor. Vulnerability Vector is extensible with different rules can be configured for various vulnerabilities. In this paper, the FileAccess and ExportedComponents Vulnerability for WebView is used as a case study.

2.1. APK Decompressor

Android Package, e.g. APK, is the package file format used by the Android mobile apps. APK files are a type of archive file, specifically in zip format-type packages. The architecture of an APK file contains all of a program's code such as .dexfiles, resources, assets, certificates, and manifest file.

A python decompressor is designed in this work, after decompression the following directories and files are acquired. In the META-INF directory, there are Manifest file (MANIFEST.MF) and certificate of the app. CERT.RSA contains the certificate of application while CERT.SF lists of all the resources with a SHA-1 digest. The lib directory contains the code at native layer with different processor architecture, such as ARM, ARMv7, ARMv8, x86, x86 64, and MIPS processor.

The res directory contains resources not compiled into resources.arsc, whilst the resources.arsc file contains precompiled resources, such as a binary XML. Assets are the directory that includes all the assets which can be retrieved by Asset Manager.

The AndroidManifest.xml is an additional Android manifest file, describing the name, version, access rights, and referenced libraries for App. It is usually in binary XML and can be converted into human-readable plaintext by using tools such as AXMLPrinter2, apktool, or Androguard. The classes.dex is executed by DVM of Android Runtime which is in dex format.
In high security or game Android Apps, packing or encryption is a prevalent protection technique. A scheme is used in the Decompressor to unpack and dump the original APK file. After that it extracts the compressed packets and gets the internal files.

2.2. Manifest.xml Parser

A manifest file contains metadata for a group of accompanying files containing descriptive information. In JAR package the manifest file specifies a version number and an entry point for execution. It may optionally contain a cryptographic hash or checksum of each file. By creating a cryptographic signature for such a manifest file, the entire contents of the distribution package can be validated for authenticity and integrity.

AndroidManifest.xml file in root directory. It declares essential information of the App, e.g. the names of Java package serves as a unique ID, components of the App (activities, services, broadcast receivers and content providers), the classes of each components and their capabilities, and the permissions of the App, etc. A parser, namely AndroidManifest.xml Parser, is designed in this work to extract corresponding information in the App for vulnerability detection. The data structure of a axml file is depicted in Fig.2. the axml file includes following fields, e.g. Magic Number, File Size, String Chunk, ResourceID Chunk, and XmlContent Chunk, etc.

![Diagram of .axml File](link)

**Figure 2.** Format of .axml File

The content of Magic Number is usually 0x00080003 with size of four bytes. The size of the App is specified in the field of File Size. The String Chunk composes Header, stringCount, styleCount, flags, stringsStart, stylesStart, and the StringPool Chunk Body. The data structure of XmlContent Chunk is depicted in the left of Fig.2 with Start Tag Trunk and corresponding End Tag Trunk.

2.3. DexParser

In the DexParser, some auxiliary engines are designed, including a Search Engine, a Filter Engine, and the Parser Engine.
After an APK is uncompressed by APK Decompressor, target string can be searched in static scanning. Thus, a Search Engine is firstly designed to accomplish target string searching according to the file structure, which is in similar with index search. Secondly, some public development classes should be filtered out to reduce analysis time, accordingly a Filter Engine is made. Finally, the specific method in the class needs to be analyzed, so the static Dex Engine is designed to analyze the vulnerability of Android App.

A Dex file contains code which is ultimately executed by the Android Runtime. Every APK has a classes.dex file, which references any classes or methods used within an app. Essentially; any Activity, Object, or Fragment used will be transformed into bytes within a Dex file. The Dex File is parsed by Dex Engine according to the file format. Dex files can be thought of as two parts. A file header which contains metadata, and a body which contains the data.

DexParser works as follows. First HeaderItem is parsed, then the Index and Data fields are analyzed. And according to this, the MapList is finally analyzed by Parser Engine in detail. The Parser Engine is designed to analyze the syntax of target Dex file. The Dalvik instruction set are divided into different categories, as depicted in Table I. The primary class constitutes Parser Engine is depicted in Fig.3.

| Opcode       | Smali bytecode  | Dalvik Instructions Categories |
|--------------|-----------------|--------------------------------|
| 0x12 <= opcode <=0x1c | [const]          | Set Register                   |
|              | [const/xx]      |                                |
|              | [const-string]  |                                |
| 0x0a<=opcode<=0x0d  | [move-result vAA] | Move values                    |
|              | [move-result-wide vAA] |                                |
|              | [move-result-object vAA] |                                |
|              | [move-exception vAA] |                                |
| 0x44 <=opcode <= 0x4a | [aget]          | Clear Register                 |
|              | [aget-xxxx]     |                                |
|              | [iget]         |                                |
|              | [iget-xxxx]    |                                |
|              | [sget]         |                                |
|              | [sget-xxxx]    |                                |
| Opcode==0x6e | [invoke-virtual] | Invoke virtual method          |

| RegisterAnalyzerVM_ImmediateValue | RegisterAnalyzerVM_ClassContainer | RegisterAnalyzerVM_Result |
|----------------------------------|----------------------------------|--------------------------|
| - init                           | - init                           | - getPath               |
| - add                            | - add_invoke_method              | - getResult             |
| - load_instructions              | - get_invoked_method_list        | - is_string             |
| - has_if_or_switch_instructions  | - get_class_name                 | - is_class_container    |
| - getInsn_return_boolean_value   | - get_class_idx                  |                          |
| - iC_class_container            |                                  |                          |
| - clear                         |                                  |                          |
| - get_stack                     |                                  |                          |
| - get_register_table            |                                  |                          |
| get_register_number_to_register_value_mapping | |                          |
| - get_register_value_by_param_in_last_index | |                          |
| - get_register_value            |                                  |                          |

**Figure 3.** Parser Engine Design
2.4. Vulnerability Vector
Vulnerability Vector is extensible and rules can be configured for different vulnerabilities, such as WebView FileAccess & ExportedComponents vulnerability, ContentProvider & Directory Traversal, Dynamically Registered Unprotected Broadcast Receiver, etc. In this work, the FileAccess & ExportedComponents vulnerability for WebView is used as a case study.

A WebView is a control to load and display web content and provide basic browser features. Control of embedded WebView is given to App via callbacks through which App can react to, modify, or reject events. A WebView may also be customized via the WebSettings class. The most important callbacks are WebChromeClient and WebViewClient respectively for browser events and web events.

In WebView, WebKit is the rendering engine. JavaScript is disabled in WebView by default, but some methods are provided to set its priority, such as setAllowFileAccess which can enable or disable file access. However, the assets and resources are still accessible even if file access is disabled. A method setAllowFileAccessFromFileURLs is provided to enable or disable JavaScript, which can be overwritten by setAllowUniversalAccessFromFileURLs.

The same-origin policy of web applications applies to WebView. Normally, Android apps run in separate processes. For instance, App A is not able to access the private data for App B, and vice versa. However, if setAllowFileAccess and setAllowUniversalAccessFromFileURLs are enabled, App A can run the exported activity from App B and pass the malicious file scheme URLs to App B to access its private files.

Take the setAllowFileAccess method as an example, its corresponding smali code is as follows:

\[
\text{-getSettings()Landroid/webkit/WebSettings setAllowFileAccess(Z)V}
\]

To interact from Javascript to Java, a Java object must be injected using the addjavascriptInterface method in webview. The following is a regular attack approach: First, a java object using addJavascriptInterface method is registered. Second, JS Injection is conducted in WebView by using webview.loadURL("javascript:XXX"). Last a JS event with JavaScript injection is delivered.

Accordingly, the addjavascriptInterface method is used in this work for WebView vulnerability detection. The smali code of addJavascriptInterface is as follows:

\[
\text{Landroid/webkit/WebView;->addJavascriptInterface()}
\]

Similarly, three typical methods are usually exploited by malicious developers to conduct remote code execution, such as searchBoxJavaBridge_, accessibility Traversal, and accessibility, which are adopted for vulnerability detection as well.

By using the StringSearch Engine, all classes including the feature code “Landroid/webkit/WebSettings” are acquired. Then the Filter Engine is applied to find all the caller methods of “Landroid/webkit/WebSettings”. Last, the ParserEngine is initiated to conduct vulnerability detection according to the corresponding smali code feature set.

3. Evaluations
In this work **Pinyin 8.16 is used as a sample to conduct a preliminary test of the Vulnerability Parser system. The detection results are categorized into Critical, Warning, Notice, and some advice is provided as well. The experimental results are depicted in Table II.
Table 2. Detection Results for **Pinyin 8.16**

| Detection Results | Description | Advice |
|-------------------|-------------|--------|
| Warning           | WebView Local File Access Attacks Checking: Found "setAllowFileAccess(true)" or not set (enabled by default) in WebView. | The attackers could inject malicious script into WebView and exploit the opportunity to access local resources. This can be mitigated or prevented by disabling local file system access. |
| Critical          | <Remote Code Execution> <#CVE-2013-4710#> WebView RCE Vulnerability Checking: Found a critical WebView "addJavascriptInterface" vulnerability. | This method can be used to allow JavaScript to control the host application. |

4. Conclusions

Nowadays, the security risk of Android application is of great importance. Android application security vulnerability and risk detection becomes an urgent research topic. In this paper, an android vulnerability analysis system is designed, namely Vulnerability Parser. It is composed of the following components, such as APK Decompressor, Manifest.xml Parser, Vulnerability Vector, and DexParser. Vulnerability Vector is extensible with various feature smali code configured for different vulnerabilities. The WebView vulnerability is used as a case study in this work, specifically the FileAccess and ExportedComponents. Experimental result demonstrates the effectiveness of the Vulnerability Parser.

5. Acknowledgment
Thanks to the staff of HuaQiang Co and the students in HIT. This work is supported by project SGSDYT00FCJS1800719.

6. References
[1] "Android Security - 2015 Year in Review," 2016.
[2] Clause J, Li W, Orso A. Dytan: A Generic Dynamic Taint Analysis Framework [J]. Proceedings of the International Symposium on Software Testing & Analysis, 2007:196–206.
[3] Wang T, Wei T, Zou W. Ro BDD-Based Fine-Grained Dynamic Taint Analysis [J]. Acta Scientiarum Naturalium Universitatis Pekinensis, 2011.
[4] C. Fritz, S. Arzt, S. Rashthofer, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau and P. McDaniel, "Highly Precise Taint Analysis for Android Application," 2013.
[5] R. Raveendranath, V. R. A. J. Babu and S. K. Datta, "Android malware attacks and countermeasures: Current and future directions." In Control, Instrumentation, Communication and Computational Technologies (ICCICCT), pp. 37-143, 2014.
[6] E. Chin, A. Porter Felt, K. Greenwood and D. Wagner, "Analyzing Inter-Application Communication in Android," in MobiSys’11, 2011.
[7] F. Wei, S. Roy and a. X. Ou, "Amandroid: A precise and general intercomponent data flow analysis framework for security vetting of android apps." Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM, pp. 1329-1341, 2014.
[8] H. Fu, Z. Zheng, S. Bose, M. Bishop and P. Mohapatra, "LeakSemantic: Identifying Abnormal Sensitive Network Transmissions in Mobile Applications." 2017.
[9] A. Continella, Y. Fratantonio, M. Lindorfer, A. Puccetti, A. Zand, C. Kruegel and a. G. Vigna, "Obfuscation-Resilient Privacy Leak Detection for Mobile Apps Through Differential Analysis," ISOC Network and Distributed System Security Symposium (NDSS), 2017.
[10] E. Mariconti, L. Onwuzurike, P. Andriotis, E. D. Cristofaro, G. Ross and A. G. Stringhini, "MAMADROID: Detecting Android Malware by Building Markov Chains of Behavioral Models," ISOC Network and Distributed System Security Symposium (NDSS), 2017.