International Journal of Computer Systems (ISSN: 2394-1065), Volume 02– Issue 09, September, 2015
Available at http://www.ijcsonline.com/

Android Platform Security Issues

Mikhail Romaneev [A] and Vartan Padaryan[B].

[A]ISP RAS, 109004, Moscow, 25 Alexander Solzhenitsyn st., melon.aerial@gmail.com.
[B]ISP RAS, 109004, Moscow, 25 Alexander Solzhenitsyn st., vartan@ispras.ru.

Abstract

In recent years mobile devices have become an integral part of daily routine, most of them based on Android OS. This leads to rapid growth of malicious applications for the Android platform, as well as specialized analysis tools. In this article we discuss current Android security issues and propose a classification of these issues by categories. Also, most common security solutions are reviewed and compared. Besides, a list of requirements for "prospective" full-system analysis solution for Android platform are proposed, as well as an architecture of such solution. The proposed solution brings together the best from the existing approaches for the analysis of Android platform, and overcomes existing weaknesses in them.

Keywords: *mobile devices, Android, software security, network security, dynamic analysis.*

I. INTRODUCTION

The use of smartphones in everyday life is not limited to voice calls and SMS. The ability to load and execute programs, as well as mobile Internet access has led to a huge number of various applications. The functionality of modern smartphone includes Internet browsers, social networking clients, office applications and applications for different kinds of Internet services. Android devices have occupied most of the smartphone market due to the open architecture of the Android platform and the convenient developer API. At the moment, Android is the most popular mobile operating system with market share of about 75%.

In recent years there has been an upsurge growth of mobile applications. According to the Gartner report [1] in 2012, 64 million applications have been downloaded worldwide, and this number has increased to 102 million in 2013. In addition to the growth of benign applications, there is also a rapid growth of malicious applications. The rapid growth of malicious applications also has generated growth in measures to counteract them. Google Play has introduced filtering of applications using Google Bouncer sandbox that analyzes malicious activity of an application. Antivirus companies have started to offer their products for Android. The number of scientific publications on this topic has increased explosively. One of the review papers [2] about security solutions for Android published in 2015 contains more than 40 projects and these were not all the known projects at that moment. Due to the fact that mobile devices displace the market of computers and laptops more and more every year, and are the source and repository of a large amount of sensitive data, security issues of Android OS and its applications are critical.

The Android platform is Open Source software. Vendors are developing their own code base and creating specialized firmware in order to achieve greater functionality and better performance. The other side of such activities is vulnerabilities and weaknesses in the implementation of algorithms that do not reside in the main code base, but on the set of existing real devices. Malware uses vulnerabilities to elevate the privileges and overcome defense mechanisms. Identification of vulnerabilities in the absence of source of firmware is extremely difficult. The primary problem is the lack of a controlled environment for its execution, which is essential for dynamic analysis.

Thus, a full analysis of device safety requires study of properties of the system and application software together. In this paper, we developed our own classification of security issues for Android platform. We also developed a list of requirements and the architecture of the "Prospective" system for full-system analysis of Android platform.

II. ANDROID ARCHITECTURE

At the core of the Android operating system is a Linux kernel with some architectural changes that were made by engineers of Google. Applications for the Android operating system are developed in Java. Starting with version Android 1.5, the Android NDK was presented, which allows to develop modules of application in C/C++ and compile them into native code [3]. Applications are available in the form of a special file format - APK, which is a ZIP-archive with a certain directory structure and files. APK-file of an application contains:

- Manifest
- Modules compiled into native code (shared .so libraries)
- Various application resources
- DEX file
- Other necessary files

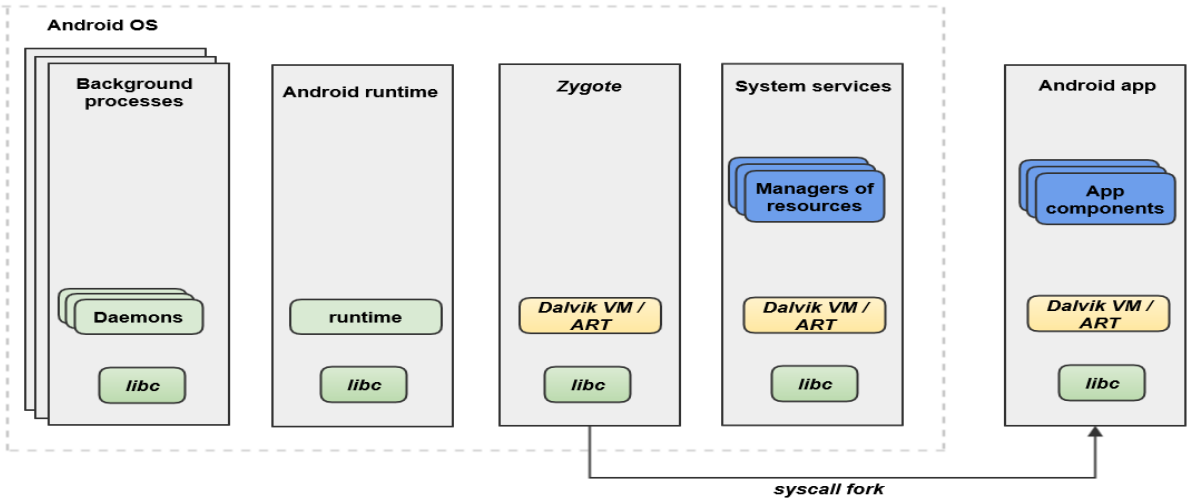

Figure 1. Start of new application in Android OS

Starting with version Android 4.4, is supported two runtime environments for applications: Dalvik VM and ART. It should be noted that the process of preparation of APK-file have not changed, the only change is in the process of installing applications in the new ART runtime. Starting with version 5.0, ART is used by default instead of the Dalvik VM.

The Dalvik VM runtime for Java programs is very different from an "ordinary" Java VM (e.g. HotSpot). Firstly, when you Java program is compiled into Dalvik bytecode, which is very different from the HotSpot JVM bytecode. Dalvik VM is register-based that makes its implementation on RISC-architectures, which are often used in mobile devices, more efficient. Memory limitations on mobile devices were also taken into account in the development of Dalvik VM. Starting with version Android 2.2, Dalvik runtime includes a JIT-compiler that compiles the "hot" pieces of code of Java programs into native code [4]. Standard Java libraries have been modified or replaced by the libraries from the Apache Harmony package, or re-written. Java programs are compiled into DEX file (Dalvik Executable), which contains the Dalvik bytecode. DEX file format has been designed to reduce the amount of memory and is significantly different from the standard format of Java programs - JAR. JNI (Java Native Interface) is used to access the modules implemented in native code. It should be noted that it is possible to load modules dynamically over the network using the component DexClassLoader [5].

III. Security Issues

The parent process for all applications in the Android OS is the Zygote process. This process is a "naked" Android application with all necessary libraries for Android environment loaded. Starting an Android application from the perspective of the operating system is as follows:

1. First fork system call creates a descendant of the process Zygote (see fig. 1).

2. This new process opens the application file (open system call).

3. Further information about class files (classes.dex) resources of the application is read from application file. Also, a socket for IPC is created.

4. Mmap system call is used to map application's files into memory.

5. The runtime environment makes required configurations and executes the application (by interpreting Dalvik bytecode or transferring control to code sections of the application in case of ART).

At the kernel level of Android OS, every application is a separate process with unique values of user/group ID, that are given to it during the installation, which runs after the launch in its own "sandbox". Starting with version 4.3, in addition to that security policy was added using of kernel components for mandatory access control - SELinux [6].

By default, the application is limited in its capabilities and cannot do anything to somehow adversely affect other applications or users: cannot read user data or modify the system applications, and has no network access. To gain access to various resources, application accesses the services of Android. Access permissions are set statically in the manifest file of the application and are issued to an application as needed while it is running. System asks the user for a consent to issue these permissions during the installation or during application updates, if they are changed. The responsibility for issuing the access to an application lies on the user, who decides to which applications give permissions for certain actions during its installation. This approach, when all permissions are issued at installation time of an application, has led to the fact that users began to distribute permissions to applications without considering the consequences. In turn, the applications began to demand more and more permissions because of the expansion of their functionality. In the currently developed version of Android 6 Marshmellow, this system of permissions is replaced by another one, where an application requests access to the resources from the user during its work, and the user can either allow access or prohibit it [7].

Android application consists of four types of components and does not contain a main() function, or any other single entry point. Application components interact with each other through special messages called Intents.

Components called Activity define the user interface. Typically, one component is used to describe a single Activity screen application. Activity can start another Activity, passing parameters through the mechanism of Intents. During operation, only one Activity may operate and process the user input, the rest of them remain frozen or destroyed, depending on the selected mode of application.

Components called Service perform background processing. When Activity needs to perform some operation, such as downloading files or playing music, and continue to work when the user interface disappears (the user switched to another application or turned it down), the application is launching a Service, the purpose of which is to perform this operation. Developers can use a Service as an application daemon which starts at boot time. Service components usually support RPC (Remote Procedure Call), so other components can access it.

Content provider maintains and exchanges data using a relational database interface. Each content provider has a unique URI for data and processes requests to it in the form of SQL queues (Select, Insert, Delete).

Broadcast receivers act as mailboxes for messages from other applications.

In papers [2, 8, 9] security issues in Android OS are discussed. Classification of those issues by category is given only in the first paper. Although the classification in the given article is very detailed and covers a lot of security issues, it has some significant drawbacks: some categories cover a wide range of vulnerabilities, while others are quite specialized; given categories of vulnerabilities do not relate to the software architecture level of Android OS; these categories haven't covered the vulnerabilities from Internet sources, and poorly covered vulnerabilities in protocol implementations and hardware (Sec. 2.7 and 2.8 in our classification). In this paper we propose the classification of vulnerabilities that eliminates these shortcomings. The categories are given below.

A. Vulnerabilities of the Linux kernel and its modules

This category of issues contains vulnerabilities that are common to all operating systems that are based on the same version of the Linux kernel as the Android OS. Exploits using vulnerabilities in the kernel can get user information or system administrator privileges. Elevating the privileges of the rights to the system administrator, the malicious software can disable the Android security system, inject malicious code into existing programs and install malware rootkits. Moreover, vendors added some kernel modules to support equipment of their devices, which may also contain vulnerabilities. Examples of working vulnerabilities of privilege escalation vulnerability are described in papers [10,11].

B. Vulnerabilities from vendor customizations

Recently, various mobile manufacturers began to produce their modified Android firmware. Such firmware usually contains a variety of applications, services and kernel components developed by the manufacturer of the device, which often could not be removed. Analysis of these modifications, described in the article [12], shows that they contain from 65% to 85% of vulnerabilities found in the entire system. In addition, it is worth noting that the vulnerabilities that have been discovered and fixed in the main branch of Android may be fixed in the branches of vendors after indefinitely long time, if fixed at all [13].

C. Vulnerabilities in modules in native code

Android applications support execution of native code through JNI (Java Native Interface). This creates another category of vulnerabilities associated with well-known memory leaks vulnerabilities in low-level languages such as C/C++ [14]. Since there are no restrictions at the process level in Android OS, except for the ones imposed by the Linux kernel, third-party libraries used in machine code may use permissions issued to the application to carry out malicious activities. Modules of application in native code also have been used by the authors of malware to avoid analysis and monitoring of Android applications. It should be noted that these modules can also be used to counter the analysis techniques used in conventional programs.

D. Vulnerabilities in inter-component communications (ICC)

This category includes vulnerabilities associated with inter-component communications between the various components of the application. Since at the level of the operating system application is limited to sandbox of the process, it needs a mechanism to access components of the Android OS and to interact with them. This occurs through the /dev/Binder device and various other services of the Android OS. Parameters of this access are specified in the manifest file in the form of an XML-file with permissions. The analysis given in the papers [8, 15, 16] indicates the shortcomings of such system of permissions and shows ways to bypass them. For example, an application can use the rights of other application to access and receive data via ICC. There may also be related to the vulnerability of third-party libraries. Third-party libraries used by the application receive the same set of constraints that the application itself. Therefore, third-party libraries may use permissions issued throughout the application to carry out malicious activities. Applications also can intercept system events sent over the broadcast request, and store information about incoming calls and SMS.

E. Vulnerabilities in applications

Each application stores some data about the user. These data should be stored properly, so other applications could not access it. For example, Skype application in one version of Android stores a database of contacts in plain form in a file on disk. Thus, contacts can be read by any other application that has access to the disk [17]. The applications can also use a cryptographic library that has errors [18] or its own implementation of cryptography, which is usually buggy. In addition, not all applications have proper user authentication and authorization. Additionally, the application may contain SQL-injections and be exposed to XSS-attacks. Most of applications are written in Java without any protection of its binary code. And as you know, Java bytecode can be easily

disassembled and analyzed. These and other vulnerabilities are described in [19, 20].

F. Vulnerabilities in Android's services and libraries

A standard set of libraries and the services running on the Android also contains vulnerabilities. For example, a vulnerability names Stagefright was recently discovered in the library that displays video of any MMS-messages, which have been exposed in all versions of Android from 2.2 [21]. Also, in following this vulnerability another vulnerability has been discovered in the MediaServer component, which affects all versions of Android from 2.3 to 5.1 [22]. The article [9] shows a vulnerability of the Dalvik runtime, where running a large number of processes in the system can lead to subsequent process running as administrator.

G. Vulnerabilities from Internet sources

Android applications are distributed across a wide number of sources in addition to the official app store. As Android applications are written mostly in Java, they are easy to reverse-engineer and repackage with malicious code [23, 24]. As it was shown in paper [25], analysis of application sandbox Google Bouncer, used in the official catalog of applications, is easy to avoid. Therefore, at the official store there is a large number of malicious programs. In addition, Android supports remote installation of applications through Google Market on devices connected to the Google account. Thus, if we hack Google account for a device, we can install a malicious application from the Google market, which was previously downloaded there. At the same time your mobile phone does not require any evidence of these actions, since they are requested in a browser window, and the application is installed in the background on your phone when you access the Internet. Also in this category there are social engineering methods, such as when to continue the work you need to install the application from unauthorized sources [26].

H. Vulnerabilities from hardware and associated with its protocols

Mobile devices running Android OS have a wide range of hardware to communicate with the outside world. These vulnerabilities can be exploited in the immediate vicinity of the device or if there is physical access to the device. Examples of these vulnerabilities are denial of service attacks on the WiFi-Direct technology, the theft of credit card data via NFC, the execution of remote code via Bluetooth, installation of malicious applications without the user's knowledge through adb via backup mechanisms and so on [27, 28, 29, 30]. Also, the article [9] shows vulnerabilities with which application privileges can be elevated using errors in the adb protocol implementation.

IV. Existing solutions for analysis of Android Applications

Since the first Android-based phone had been released, a large number of tools for the analysis of Android-applications have been written. The most comprehensive review of these instruments is presented in the papers [31, 32, 33, 2]. In the first three articles these tools are classified by the type of analysis used in them: static, dynamic, and their union (mixed). The article [2] used stages of deployment of an application on Android device for classification of tools. In this paper we use the first approach for classification of these tools.

A. Static analysis tools

Static analysis tools can be divided into the following categories:

- Metadata extraction: tools that extract information from the application manifest and provide information about requested permissions, activity components, services and registered broadcast receivers. Meta information is often used later in the dynamic analysis functions to run the application.
- Instrumentation: tools for modifying existing bytecode of applications using the technique of instrumentation. Using this technique can be added such as tracing functionality to existing applications.
- Decompiler: tools that implement decompiler and disassembler for Dalvik bytecode.

One of the most popular tools for reverse engineering of Android apps is Apktool [34]. It has the ability to decode application resources to about original form, repackaging of applications back into the APK/JAR files, debugging of smali bytecode. For decompiling and compiling of Dalvik bytecode Apktool uses another renowned project smali/backsmali [35]. Also, a tool named Dedexer is often used for disassembling Dalvik bytecode [36].

Radare2 [37] is an open source tool to disassemble, analyze, debug and modify binary files of Android applications.

One of the most comprehensive tools for static analysis is Androguard [38]. It can decompile and disassemble the application back to Java source code. If you give it two APK files it can find the coefficient of similarity and detect repackaging of application or known malicious applications. Due to its flexibility it is used in some instruments of mixed analysis.

Given static analysis tools are not all that exist at the moment. A more complete list can be found in the articles

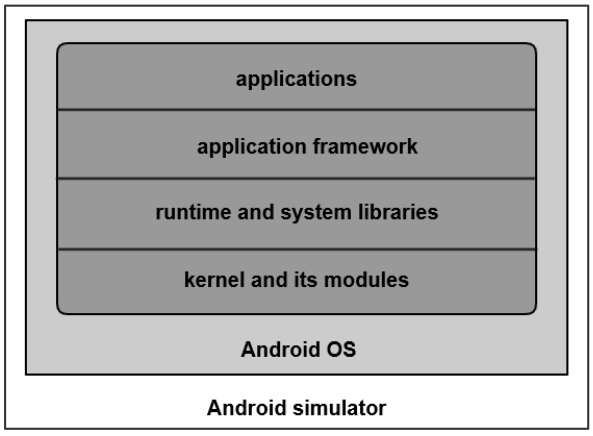

Figure 2. Architecture levels of Android sandbox.

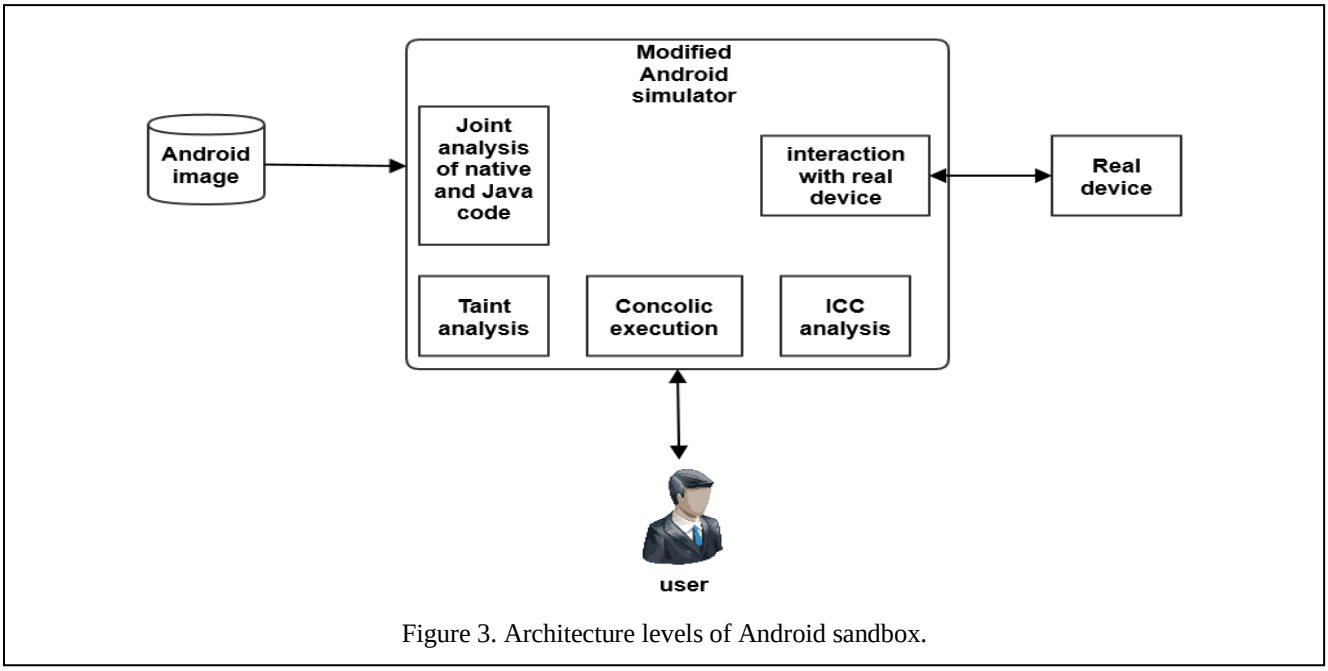

Figure 3. Architecture levels of Android sandbox.

given above.

It should be noted that although static analysis tools on the one hand can cover the entire program, on the other hand, static analysis has a number of limitations associated with the fact that it has only an abstract representation of the program. For example, static analysis becomes much more difficult if some obfuscations techniques are applied to the program. Depending on the difficulty level of obfuscation, some and perhaps all the static approaches may be absolutely useless. If the call of each method is made only indirectly, it is unlikely for these tools to build a call graph of the program. On Android, this can be done using the Java Reflection API to call methods and create objects instead of using ordinary calls and creating objects with 'new' instructions. The market has already presented several solutions for the protection of the source code files by obfuscating Android applications that can negate the benefits of static analysis [39, 40]. More information about techniques to counter static analysis can be found in the paper [41].

B. Dynamic/mixed analysis tools

Dynamic analysis tools monitor the behavior of unknown applications at runtime by running it in a controlled "sandbox" for getting its behavioral track. For getting this track they use instrumentation of sandbox on its different architecture levels with code sections that monitor the behavior of the application and Android OS. These levels are shown on figure 2.

As shown in Figure 2, architecture of the "sandbox" is an Android emulator (usually QEMU) inside which Android OS runs. Instrumentation is made either at the level of the emulator or at the level of Android operating system, or both. The level of the Android OS is also divided into 4 sublevels:

- applications
- application framework
- run-time and libraries
- kernel and its modules

Techniques used in the dynamic analysis for obtaining behavioral track are listed below:

- Taint analysis: Tools for tracking tainted data are often used in dynamic analysis for the implementation of entire system analysis of the tainted data to monitor potential leaks of confidential information.
- Monitoring the system calls: Tools can record system calls using instrumented virtual machine, strace or kernel module. This allows tracing of the machine code.
- Tracing of methods: Tools can keep track calls of Java methods of an application in Dalvik VM, calls of functions in machine code through JNI, system calls and interrupts.

Mixed analysis tools are such tools that combine the technologies of dynamic and static analysis..

V. "PROSPECTIVE" SYSTEM FOR FULL-SYSTEM ANALYSIS OF ANDROID PLATFORM

Articles [2, 31, 32, 33] describe over 40 different tools for analyzing Android apps and for analysis of the Android OS in general. As noted in the articles [25, 42, 43], existing dynamic analysis tools have a number of drawbacks. It should be noted that these drawbacks are also inherent in the standard analysis tool apps in Google Play store – Google Bouncer, as a result, a malicious application may be distributed through the official app store without being detected.

A comparison of the capabilities of approaches and systems in the reviewed publications enable us to formulate an architecture and a list of requirements for "prospective" system for full-system analysis of Android platform, which is able to analyze the application and all layers of system software together. This system borrows some effective techniques from the reviewed previous work, combining

them with the aim to overcome the drawbacks of existing solutions.

The system architecture is presented in fig. 3. The main component is a full system simulator inside which runs as an unmodified Android OS image for simulator, as unmodified images of Android OS from real devices. The simulator supports forwarding sensors from real devices, and execution of individual sections of native code on real devices. Internally, the simulator consists of the following components:

- Concolic execution component
- Tracing component
- Taint analysis component
- ICC analysis component
- Component for interaction with real devices
- Component for joint analysis of native and Java code

Here is a developed list of requirements for this system and its components:

A. Support for loading Android images from vendors into the simulator

A major shortcoming of all existing analysis tools of Android and its applications is the inability to run Android OS images from vendors in the simulator. As it was mentioned in paragraph 2.2., vendor customizations of Android contain from 65% to 85% of the vulnerabilities found in the whole system. At the moment, there are no tools for analysis which can load Android images from vendors. All existing solutions work on a special simulator build of Android OS.

B. Possibility of execution of individual pieces of native code on a real device

According to information from the articles [25, 42], there are ways to identify execution of Android OS inside the simulator. Usually, techniques for detecting execution in the emulator are based on identification of working QEMU binary translation mechanism, because QEMU is the simulator on which most of Android sandboxes are based. These techniques are based on differences in behavior of some native code inside simulator and on real device. Since changing of working mechanism of binary translation is a complex task, running individual pieces of native code on a real device would be a good approach to confront these methods of detection simulator.

C. Forwarding data from sensors of real devices in the simulator

As described in articles given in preceding paragraphs there are ways to detect simulators based on data received from sensors of the device hardware, such as: accelerometer, gyroscope, GPS, light sensor, gravity. The output values of sensors are based on information gathered from the environment, and therefore their realistic simulation is a complex task. The presence of this kind of sensors is the main difference between smartphones and desktop computers. The increasing number of sensors on smartphones provides new opportunities to identify mobile devices.

D. Joint analysis at the level of Java code and native code

The difficulty for many systems for analysis of Android applications is that applications contain both modules in Dalvik bytecode and in native code. Among existing tool, support for analysis of all modules is implemented only in DroidScope [44] and CopperDroid [45]. «Prospective» system for full-system analysis should allow monitoring data flow and control flow at user and system levels. For user code, when it is possible, level of execution should be raised to Java code, which is a high-level programming language. It is also necessary to support integration of data flows and control flows from native code to Java code and vice versa.

E. Support analysis of ICC

The article about CopperDroid given in preceding paragraph described support for analysis of inter-component communication both inside an Android application and between different applications. CopperDroid does this by intercepting data passing through core component Binder, since all communication goes through it. The approach implemented in CopperDroid allows to not make modifications of Android source code, which makes it possible to port it to the new version of Android OS and new ART run-time with minimal changes.

F. Protection from static detection heuristics

As shown in the articles [25, 42], most of the tools for dynamic analysis can be detected by simply checking values of IMEI, IMSI, or number of build of the Android image. Also, simulator can be detected by checking the standard values of routing table. The only sandbox that has protection against detection with static heuristics is ApkAnalyzer [46].

G. Minimal Android image modifications

It is also worth noting that many of tools for dynamic analysis are based on instrumentation of code of various components of Android OS, particularly virtual machine, DalvikVM. This complicates their continued support, as well as migration to the new ART run-time.

H. Support of full-system taint analysis with tracking implicit control flows

It is worth noting that many of the tools for dynamic analysis using supporting of taint analysis with implantation from the tool TaintDroid. The article [43] shows successful ways to circumvent the analysis of that tool. The reason of success of these attacks are the following facts:

1) TaintDroid only tracks data flows (explicit flows) and doesn't track control flows (implicit flows).

2) TaintDroid monitors data flows only at the level of Dalvik VM.

Possible ways to resolve these shortcomings are described in the [43] and are directions for further researches.

I. Support of concolic execution

The article [47] describes implementation of concolic execution. Concolic execution combines static analysis techniques for analyzing control flow graph of program, symbolic execution of program and execution of program with specific values. This approach is described in all details in papers [48, 49]. The aim of Condroid is observation of execution paths that lead to the sections of the program containing "interesting" code. By "interesting" code we mean code, execution of which depends on occurrence of any external event or the system environment settings. For example, dynamically loadable class code in Android with the help of DexClassLoader component, or calling native methods through JNI.

VI. Conclusion

In this article we have surveyed Android security issues. We have proposed our classification of these issues by categories. We have also compared and reviewed most well-known security solutions for Android platform. Our survey additionally shows the advantages and limitations of these solutions. We also have proposed current directions for research in this area. Besides, we have developed a list of requirements for "prospective" full-system analysis solution for Android platform, as well as the architecture of the solution.

Acknowledgment

This work is supported by RFBR, grant 15-07-07652.

References

[1] I. Lunden. Gartner: 102b app store downloads globally in 2013, 26b in sales, 17% from in-app purchases, http://techcrunch.com/2013/09/19/gartner-102b-app-store-downloads-globally-in-2013-26b-in-sales-17-from-in-app-purchases/

[2] Tan, D. J., Chua, T. W., & Thing, V. L. (2015). Securing Android: A Survey, Taxonomy, and Challenges. ACM Computing Surveys (CSUR), 47(4), 58.

[3] Android Developer Documentation: Android NDK Native APIs, http://developer.android.com/ndk/guides/stable_apis.html .

[4] Dalvik VM Internals, https://sites.google.com/site/io/dalvik-vm-internals

[5] Android Developer Documentation: DexClassLoader class, http://developer.android.com/reference/dalvik/system/DexClassLoader.html

[6] SELinux in Android, https://source.android.com/devices/tech/security/selinux/

[7] Android Developer Documentation: Android M Permissions, https://developer.android.com/preview/features/runtime-permissions.html

[8] Enck, W., Ongtang, M., & McDaniel, P. (2009). Understanding android security. IEEE security & privacy, (1), 50-57.

[9] Shabtai, A., Mimran, D., & Elovici, Y. (2015). Evaluation of Security Solutions for Android Systems. arXiv preprint arXiv:1502.04870.

[10] Hei, X., Du, X., & Lin, S. (2013, June). Two vulnerabilities in Android OS kernel. In Communications (ICC), 2013 IEEE International Conference on (pp. 6123-6127). IEEE.

[11] Zhou, X., Demetriou, S., He, D., Naveed, M., Pan, X., Wang, X., ... & Nahrstedt, K. (2013, November). Identity, location, disease and more: Inferring your secrets from android public resources. In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security (pp. 1017-1028). ACM.

[12] Wu, L., Grace, M., Zhou, Y., Wu, C., & Jiang, X. (2013, November). The impact of vendor customizations on android security. In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security (pp. 623-634). ACM.

[13] Zhou, X., Lee, Y., Zhang, N., Naveed, M., & Wang, X. (2014, May). The peril of fragmentation: Security hazards in android device driver customizations. In Security and Privacy (SP), 2014 IEEE Symposium on (pp. 409-423). IEEE.

[14] Sun, M., & Tan, G. (2014, July). NativeGuard: Protecting android applications from third-party native libraries. In Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks (pp. 165-176). ACM.

[15] Octeau, D., McDaniel, P., Jha, S., Bartel, A., Bodden, E., Klein, J., & Le Traon, Y. (2013). Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In USENIX Security 2013.

[16] Chin, E., Felt, A. P., Greenwood, K., & Wagner, D. (2011, June). Analyzing inter-application communication in Android. In Proceedings of the 9th international conference on Mobile systems, applications, and services (pp. 239-252). ACM.

[17] CVE-2011-1717, http://www.cvedetails.com/cve/CVE-2011-1717/

[18] Fahl, S., Harbach, M., Muders, T., Baumgärtner, L., Freisleben, B., & Smith, M. (2012, October). Why Eve and Mallory love Android: An analysis of Android SSL (in) security. In Proceedings of the 2012 ACM conference on Computer and communications security (pp. 50-61). ACM.

[19] Projects/OWASP Mobile Security Project - Top Ten Mobile Risks, https://www.owasp.org/index.php/Projects/OWASP_Mobile_Security_Project_-_Top_Ten_Mobile_Risks

[20] Lu, L., Li, Z., Wu, Z., Lee, W., & Jiang, G. (2012, October). Chex: statically vetting android apps for component hijacking vulnerabilities. In Proceedings of the 2012 ACM conference on Computer and communications security (pp. 229-240). ACM.

[21] Android Stagefright vulnerabilities, https://www.kb.cert.org/vuls/id/924951

[22] CVE-2015-3842, http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3842

[23] Zhou, Y., Wang, Z., Zhou, W., & Jiang, X. (2012, February). Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In NDSS.

[24] Nolan, G. (2012). Decompiling android. Apress.

[25] Petsas, T., Voyatzis, G., Athanasopoulos, E., Polychronakis, M., & Ioannidis, S. (2014, April). Rage against the virtual machine: hindering dynamic analysis of android malware. In Proceedings of the Seventh European Workshop on System Security (p. 5). ACM.

[26] Android Security Underpinnings, http://www.youtube.com/watch?v=NS46492qyJ8

[27] Android WiFi-Direct Denial of Service, http://www.coresecurity.com/advisories/android-wifi-direct-denial-service

[28] NFC attack can steal your credit card information, http://securityaffairs.co/wordpress/37667/hacking/nfc-attack-credit-card.html

[29] Google Android Bluetooth Forced Pairing Vulnerability, http://www.zerodayinitiative.com/advisories/ZDI-15-092/

[30] Android ADB backup APK injection vulnerability, http://www.securityfocus.com/archive/1/535980/30/150/threaded

[31] Neuner, S., van der Veen, V., Lindorfer, M., Huber, M., Merzdovnik, G., Mulazzani, M., & Weippl, E. (2014). Enter sandbox: Android sandbox comparison. arXiv preprint arXiv:1410.7749.

[32] Faruki, P., Bharmal, A., Laxmi, V., Ganmoor, V., Gaur, M., Conti, M., & Muttukrishnan, R. Android Security: A Survey of Issues, Malware Penetration and Defenses.

[33] Hoffmann, J. (2014). From Mobile to Security: Towards Secure Smartphones (Doctoral dissertation).

[34] ApkTool, http://ibotpeaches.github.io/Apktool/

[35] Smali, https://github.com/JesusFreke/smali

[36] Dedexer, http://dedexer.sourceforge.net/

[37] Radare2, http://www.radare.org/r/

[38] Androguard, https://github.com/androguard/androguard

[39] DexProtector, https://dexprotector.com/

[40] DexGuard, https://www.guardsquare.com/dexguard

[41] Protsenko, M., & Muller, T. (2013, October). PANDORA applies non-deterministic obfuscation randomly to Android. In Malicious and Unwanted Software:" The Americas"(MALWARE), 2013 8th International Conference on (pp. 59-67). IEEE.

[42] Jing, Y., Zhao, Z., Ahn, G. J., & Hu, H. (2014, December). Morpheus: automatically generating heuristics to detect Android emulators. In Proceedings of the 30th Annual Computer Security Applications Conference (pp. 216-225). ACM.

[43] Sarwar, G., Mehani, O., Boreli, R., & Kaafar, M. A. (2013, July). On the Effectiveness of Dynamic Taint Analysis for Protecting against Private Information Leaks on Android-based Devices. In SECRYPT (pp. 461-468).

[44] Yan, L. K., & Yin, H. (2012, August). DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In USENIX security symposium (pp. 569-584).

[45] Tam, K., Khan, S. J., Fattori, A., & Cavallaro, L. (2015). CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In Proc. of the Symposium on Network and Distributed System Security (NDSS).

[46] ApkAnalyzer, https://www.apk-analyzer.net/

[47] Schütte, J., Fedler, R., & Titze, D. (2014). Condroid: Targeted dynamic analysis of android applications. in review.

[48] Sen, K. (2009, October). DART: Directed Automated Random Testing. In Haifa Verification Conference (Vol. 6405, p. 4).

[49] Sen, K., Marinov, D., & Agha, G. (2005). CUTE: a concolic unit testing engine for C (Vol. 30, No. 5, pp. 263-272). ACM.