Android Malware Analysis
Based On Memory Forensics

András Gazdag and Levente Buttyán

Laboratory of Cryptography and System Security (CrySyS Laboratory)
Budapest University of Technology and Economics
`{andras.gazdag,buttyan}@crysys.hu`
`http://www.crysys.hu`

Abstract. Live forensics solutions have long been proven powerful in various research fields. The rise of mobile platforms has created numerous new challenges for the researchers. The adoption of the widely used technologies of the traditional PC environment has limitations due to the lack of wider control over the mobile operating system. In this paper we present a new malware analysis solution for the Android platform using a memory forensics approach. We explore the required modification of the Android system to be able to use it as a memory analysis environment and demonstrate the solution with an implementation. We propose possible analysis targets for the acquired memory image. Based on the information gathered from the analysis steps we present a methodology of behaviour analysis of android applications, furthermore we show the power of this approach analysing and evaluating Android malwares. Finally we evaluate our implementation with well-known malware family samples illustrating its efficiency and effectiveness.

Keywords: Android; malware; dynamic; analysis; memory; forensics

1 Introduction

In the last couple of years we saw a dramatic increase in the mobile malware area. F-Secure shows in its latest Mobile Threat Report[1] that more than 99% of the new malware samples found in the wild target the Android ecosystem. Most of these samples (83%) are still Trojan applications followed by Backdoor applications. Hidden SMS sending to premium services is still the most common type of malicious behaviour.

During our research we used a memory forensics approach for analysing mobile devices and detecting malware. This technique is similar to the traditional memory forensics in the desktop environment which was first introduced by Michael Ford in 2004[2]. The goal of memory forensics is to capture the contents of the RAM to a memory image, which is later analysed in various ways to gain information about the system that is otherwise not accessible. Mobile memory forensics was first introduced by Joe Sylve in 2012[3].

We designed and implemented a system containing two components to detect malicious Android applications. First we present UkatemiSHIELD, an Android application, aiming to find suspicious applications based on their requested permissions. Suspicious applications can then be analysed with another component of our system called APKAnalyser, which is a memory forensic tool for Android. We performed extensive testing to prove the robustness of our approach.

The reminder of this paper is organised as follows. Section 2 gives an overview of the implemented system then section 3 and 4 describe the structure and operation of the two components of the system: UkatemiSHIELD and APKAnalyser. Finally, section 5 describes that APKAnalyzer can be used to detect malicious behaviours and discusses the results.

2 System Overview

The complete system contains two components: UkatemiSHIELD and APKAnalyser. These components are capable of operating separately, but their cooperation can result in a more robust defence system. Figure 1. shows the overview of the system.

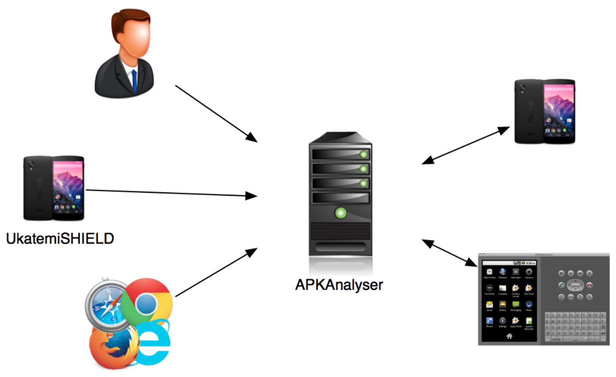


Fig. 1. The complete system overview of UkatemiSHILED and APKAnalyser.

2.1 UkatemiSHIELD

UkatemiSHIELD is an Android application with the goal of defending and analysing the underlying Android system. It runs periodic checks of the environment searching for compromised files or malicious applications. If it finds a suspicious file or application, then it generates an alert, and sends the available suspicious files for further analysis to APKAnalyser.

2.2 APKAnalyser

APKAnalyser is a web service that analyses Android applications in a .apk format. It performs multiple automated analysis steps, and produces a report that helps the work of human researchers. First, it searches the VirusTotal database to check whether the submitted sample is an already known malware. Afterwards it performs dynamic analysis on the file, and records its behavioural features.

3 UkatemiSHIELD

During the development of UkatemiSHIELD, an important objective was to stay within the boundaries defined by Google. This approach results in an application available for as many users as possible. On the other hand, this also restricts the possibilities of the application, reducing the level of protection.

UkatemiSHIELD performs a system-wide file scanning. It calculates hashes on every file, and compares them to a list of trusted file hashes.[4] This approach is particularly powerful on the system partition, where during a normal use of the device no modification should happen. Any discrepancy on this partition is probably the result of malicious activity.

The software performs another type of analysis as well. It checks all the installed applications on the system reviewing their permission request. Based on this information, it makes an automated decision, using machine learning, whether the application is suspicious or not. It has been shown[5] that this methodology is useful detecting Android malware. If an applications is marked as suspicious, then it is sent to the APKAnalyser service for further analysis.

4 APKAnalyser

APKAnalyser is a web service for analysing Android applications. It has two goals: based on various performed steps it decides whether the submitted application has malicious behaviour, and it generates an analysis report to ease the work of human researchers.

APKAnalyser has multiple user interfaces for more convenient usage. It offers a web interface through which users can submit files for analysis. It also has a REST based file upload API which is used for communications with programs, such as UkatemiSHIELD. Local users of the service have the option of adding multiple files at once from the file system allowing them to perform batch analysis.

4.1 Analysis overview

The analysis runs on an environment specifically modified for memory analysis purposes. We use the LiME[3] module to capture memory images. The images are processed then with the Volatility tool[6].

4.2 Analysis environment

One of the main design decisions of APKAnalyser was to create a flexible environment for multiple purposes. The software was developed in such a way that it is capable of operating with emulated Android devices, but it also has the option to perform all the steps on a physical device. This approach gives the researchers the options to work with a highly scalable cost effective solution or to work with an environment that is as close to a real device as possible.

LiME is an LKM (Loadable Kernel Module). It needs to be loaded into the Android kernel which is by default not permitted due to security reasons. Therefore, we needed to compile custom kernels with the required functions enabled.

The new kernel image can be used with Android emulator without any further modification.

However, the usage of a custom kernel with a physical device is more complicated. First the bootloader of the device has to be unlocked. The mechanism for this depends highly on the vendor. We performed the steps on a Sony Xperia Arc S device which was easier than with other brands thanks to the support from Sony for their developers. The unlocked bootloader allowed us to load a new boot image with our custom kernel on the device. It is possible to use the fastboot tool, which allows the user to load the boot.img into the memory of the device without any modification of the permanent storage. This approach is useful, because in case of an error, no permanent damage is made to the device.

LiME is capable of running in two modes: it can store the captured memory image on the SD card of the device or it can send out the contents of the image via a TCP connection. We used it in the second mode because it suited better our environment.

Apart from a custom kernel, root privileges are also required for loading the LiME module. It can be achieved in various ways. In the emulated environment, we used a custom su binary copied on the system partition, which runs then with root privileges. We developed a custom version of the Superuser.apk[1] application which handles the root privilege request from other applications to grant the root privileges for our application automatically. This modification was required to be able to automate the complete analysis process.

Gaining root privileges is also required on physical devices. Methodology for this depends highly on the device.

4.3 Memory image analysis

We analysed the captured memory image with Volatilty. Android at the kernel level shows no significant difference from a Linux system, hence all the Linux commands can be used to analyse Android memory images.

In order to use Volatility, we needed to have a new Volatility profile set up that describes the structure of the memory image allowing Volatility to analyse its content.

The available Volatility commands allowed us to gather informations about running processes, open network connections, open file descriptors or modifications of the kernel structures.

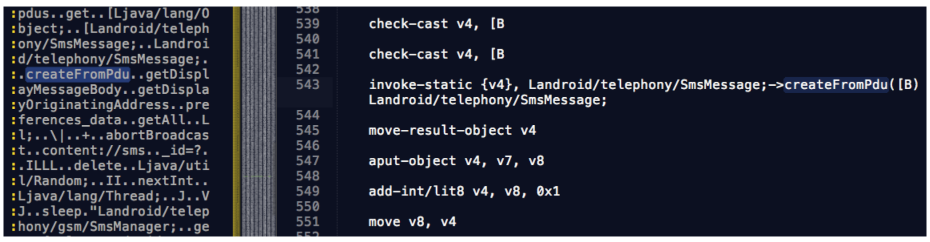
[1] We obtained the su binary and Superuser.apk from the XDA developer forums.[7]

We examined the running processes with the following commands: linux_pslist, linux_pstree, linux_psaux [-p pid], linux_proc_maps [-p pid], linux_dump_map [-s address].

Finding the process. As a first step of the memory image analysis we had to find the process (and the associated process id) containing the analysed application. We could achieve this in two different ways. In the first approach, there were only known applications running in the analysis environment, which allowed for an easy filtering of known processes. The second approach allowed us to run any applications in the analysis environment. In this case we captured two memory images: before and after the installation of the sample. By comparing the two images, we could again filter out the irrelevant processes.

Analysing the memory map. With the known process id (pid) we could get the memory map of the analysed process. Every Android application has a lot of binaries and further apk-s loaded into their memory map in order to run as fast as possible. The interesting memory page for this analysis is the one that contains the Dalvik byte code of the investigated application. This page is a `READ ONLY` page as the code will actually be executed by the Dalvik Virtual Machine in another memory segment.

Dumping memory. Dumping specific parts of the memory can give us back the Dalvik executable (dex) of the analysed application. It contains Dalvik byte code which is readable for humans as well. Further analysis of this dex file can reveal important informations about the behaviour of the program. Figure 2. shows a comparison of the dumped dex file with original dex file of the application.


```
:pdus..get..[Ljava/lang/O
:bject;..[Landroid/teleph
:ony/SmsMessage;..Landroi
:d/telephony/SmsMessage;.
:.createFromPdu..getDispl
:ayMessageBody..getDispla
:yOriginatingAddress..pre
:ferences_data..getAll..L
:l;..\|..+..abortBroadcas
:t..content://sms.._id=?.
:.ILLL..delete..Ljava/uti
:l/Random;..II..nextInt..
:Ljava/lang/Thread;..J..V
:J..sleep."Landroid/telep
:hony/gsm/SmsManager;..ge
```

```
check-cast v4, [B

check-cast v4, [B

invoke-static {v4}, Landroid/telephony/SmsMessage;->createFromPdu([B)
Landroid/telephony/SmsMessage;

move-result-object v4

aput-object v4, v7, v8

add-int/lit8 v4, v8, 0x1

move v8, v4
```

Fig. 2. Comparison of the dumped dex file with the original dex file of the application.

Analysis with other goals. Volatility allows researchers to analyse the memory image with other goals. It is possible to regain information about the tempfs file system that is used as a caching partition on Android systems. Discovering the contents of this file system can give us information for example about the last opened URLs.

Volatility also has mechanisms that can be used for finding rootkits in Android systems. The command linux_psxview gathers information about running processes from various sources from the kernel and cross references this information. Any discrepancy found means that a program tried to hide itself. This is a frequently encountered malicious behaviour. Checking the outputs of commands like linux_check_fop or linux_check_afinfo or linux_check_modules can result finding further attempts of hiding files, network connections or kernel modules.

5 Determining application behaviour

APKAnalyser is capable of making automated decisions about application behaviour. It performs a number of analysis steps described in the previous section: it sets up an analysis environment, installs the analysed sample and performs memory capture of the analysis device. It then searches for memory pages in the captured memory image containing the program code. Once it has dumped the dex files from the image, it analyses them by collecting system calls. Based on the system calls performed by the application it distinguishes malicious applications from the legitimate ones.

5.1 System call list

It is essential for proper functioning of APKAnalyser to have a list of system calls to look after. If the system call list contains calls specific to malwares then it is capable of distinguishing malicious files and legitimate ones.

A more specific selection of system calls could result in a system that can recognise samples of the same malware family. During our testing we showed that both of these approaches work.

For testing purposes, we created a list of system calls that is specific to malwares in general. We used the information gathered during the development of UkatemiSHILED: we took the most relevant permissions that malwares use and searched for the corresponding system calls representing that permission usage. A system call based analysis can be much more detailed than a permission based, so we extended further this list. For example the most common malware behaviour is SMS sending. We included in the list all methods of the SmsManager[2] class that can be associated with creating and sending new messages.

The relevance of system calls may vary in the list. Some calls are more characteristic to malwares whereas some may also be frequent among legitimate applications. To represent this relevance, we assigned a weight to each system call that describes the impact of the presence of a particular system call on the behaviour score.

[2] The SmsManager class description can be found here: `http://developer.android.com/reference/android/telephony/SmsManager.html`

After the system call list is completed, a threshold should be determined. If the behaviour score for a sample is higher than the threshold then the sample should be treated as a malware. We performed extensive testing on malicious and legitimate applications to determine the correct threshold value for a selected system call list.

5.2 Testing

For testing the completed system, we have chosen malware samples from various sources. We checked the samples against the VirusTotal database to verify the decision made by APKAnalyser. We removed those samples from the measurement that crashed during their run in the environment.

This has lead to a total of 111 tests. The results show a 71,4% true positive rate and a 26,7% false positive rate. The rates depend highly on the chosen system call list and the corresponding threshold value. The system call list we chose is described in section 5.1. The correct threshold value and weighting of the system calls were determined with extensive testing.

We have performed tests to prove the malware family detection capability as well. We chose samples from the DroidKungFu (HEUR: Backdoor.AndroidOS. KungFu.a) family and the Lotoor (Exploit.Linux.Lotoor.b) family.[3] We performed the test two times: once the system call list was customised for DroidKungFu and once for Lotoor. We could make 25 tests in total with samples from these families. We were able to identify samples with 80% probability from the DroidKungFu family and with 93,3% probability from the Lotoor family.

6 Related work

The foundations for complete mobile memory forensics came from the work of Joe Sylve et al.[8]. In his work he presented LiME[3] and showed that it is capable of capturing memory pages with higher accuracy than any other approach. However, the authors did not analyse further the memory dumps to discover informations about the running applications.

Joe Sylve and co-workers continued their work at 504ensics where they work on developing Dalvik Inspector (DI)[9]. The goal of this tool is to analyse Dalvik level structures of programs from memory images. This application is currently under development.

Christos Xenakis et al. investigated whether authentication credentials in the volatile memory of Android mobile devices can be discovered using freely available tools or not[10, 11]. The authors used DDMS for memory acquisition which only captures a part of the system memory. They found out that most of the mobile banking, e-shopping and password manager applications keep the credentials informations in memory long after it is necessary. They concluded that with the memory forensics approach it may be possible for malicious applications to stole credential informations.

[3] These naming conventions follow the Kaspersky Android malware classification terminology.

7 Summary

In this paper we proposed UkatemiSHIELD and APKAnalyser. These new tools form an analysis system to detect malicious Android applications. We showed that APKAnalyser in cooperation with UkatemiSHIELD is capable to detect malicious behaviour using a memory forensics approach.

In order to validate our method we performed 136 tests with different application samples. Our tests showed that the selection of the system calls is crucial for the effective operation of APKAnalyser. With careful choice of the system call list, the corresponding weight values and a proper threshold value, APKAnalyser is able to detect malwares with high confidence. Tests also proved that this methodology is particularly powerful for detecting samples of a specific malware family.

Acknowledgments. The authors extend thanks to Szabolcs Váradi for his work on UkatemiSHIELD and Gábor Pék who contributed greatly to this project.

References

1. F-Secure Labs: Mobile Threat Report Q1 2014 `http://www.f-secure.com/documents/996508/1030743/Mobile_Threat_Report_Q1_2014.pdf`
2. Michael Ford: Linux Memory Forensics `http://www.drdobbs.com/linux-memory-forensics/199101801`
3. Joe Sylve: Android Mind Reading: Memory Acquisition and Analysis with DMD and Volatility (2012)
4. Kim, Gene H. and Spafford, Eugene H., "The Design and Implementation of Tripwire: A File System Integrity Checker" (1993). Computer Science Technical Reports. Paper 1084.
5. Borja Sanz and Igor Santo and Carlos Laorden and Xabier Ugarte-Pedrero and Pablo Garcia Bringas and Gonzalo Álvarez: PUMA: Permission Usage to detect Malware in Android (2012)
6. AAron Walters and Nick L. Petroni, Jr.: Volatools: Integrating Volatile Memory Forensics into the Digital Investigation Process (2007)
7. xdadevelopers: [How-To] Root an Android Virtual Device (AVD) `http://forum.xda-developers.com/showthread.php?t=2227815`
8. Joe Sylve, Andrew Case, Lodovico Marziale, Golden G. Richard: Acquisition and Analysis of Volatile Memory from Android Devices, Digital Investigation Journal, 2012
9. 504ensics: Dalvik Inspector (DI) Alpha `http://www.504ensics.com/tools/dalvik-inspector-di-alpha/`
10. Dimitris Apostolopoulos, Giannis Marinakis, Christoforos Ntantogian, Christos Xenakis, "Discovering authentication credentials in volatile memory of Android mobile devices", In Proc. 12th IFIP Conference on e- Business, e-Services, e-Society (I3E 2013), Athens, Greece, April 2013.
11. Christoforos Ntantogian, Dimitris Apostolopoulos, Giannis Marinakis, Christos Xenakis, Evaluating the privacy of Android mobile applications under forensic analysis, Computers & Security, Elsevier Science, 2013.