IJST International Journal of Science and Technology
CENTRE OF PROFESSIONAL RESEARCH PUBLICATIONS
ISSN 2049-7318

International Journal of Science and Technology Volume 2 No. 9, September, 2013

System Calls Analysis of Malwares on Android

F. Tchakounté, P. Dayang
The University of Ngaoundéré, Faculty of Science,
Department of Mathematics and Computer Science, PO Box 454 Ngaoundéré, Cameroon

ABSTRACT

Android devices are targeted by malicious developers whose aim is to infiltrate these equipments in order to manipulate user's data. They insert malicious code inside the applications they publish into Google Play or unofficial app-stores. Once installed, these apps disguise and send back information to the attacker's server. In general, mobile malware can be analysed using two different and complementary techniques: static and dynamic analysis. The first one consists on scrutinizing behaviour of malware during execution and the second consists on the reverse engineering of the malicious application from the .apk and mainly focuses on AndroidManifest.xml and classes.dex files. In this paper, we scrutinize dynamic low-level analysing kernel invocations initiated by the malicious code at the moment the user runs it. Most often, the attacker entices the user in presenting him well designed User Interface (UI) to convince him to apply naively an action. These interfaces can be to validate a form, check, select, click a button or to open a window. In this work, we found a new scenario of how the user can be lured to aid the malicious developer. We discovered that if the user clicks in a region of mobile screen that is distinct to what the attacker programmed to lure the user, attacker malicious events are triggered. So, the user is continuing to participate unintentionally with random events (such as click on the activity window) to the malicious spreading of the malware inside the system. This result show firstly that the user does not need to manipulate (interact with) the application to divulge unconsciously its sensitive information and secondly that the system lacks to control events initiated by the user on applications unintentionally. Moreover, we confirm that malwares may not start automatically at the Kernel layer: they require the user to manually run the infected application.

Key words: *Malware, analysis, Android, User Interface, malicious, unconscious*

1. INTRODUCTION

Android is the mobile operating system widely used in the world [6][7]. It is freely available but represents a target for attackers that focus their attention on the platform [8]. Android is seriously affected by a variety of security threats. Below are some common threats, which explicitly need the user participation as well as techniques they use to infiltrate and spread [9][25].

Repackaging: Attackers take a benign application, modifies it with malicious code and republishes it into app-store. Then users can download without detecting any problems.

Update attacks: Attackers disclose an interesting benign application into Google Play to gain many users. Once done, this application is updated with malicious code. All users then will be infected as soon as they manually or automatically update their apps.

Misleading disclosure: attackers disclose their functionality in a user interface well presented to convince the user to interact with malicious code.

Phishing scams: application that entices people via user interfaces, Web pages or mails to provide its information to a malicious server controlled by attackers.

Drive by downloads: it refers to downloads launched automatically when a user is on a Website. They use spams or malvertising to bring users to Website that automatically launches a fake download. It consists on convincing the user to perform an action to open the downloaded application.

Many users are not aware of attacker techniques; they help unconsciously the attacker to perform malicious actions. As many users are not experts, they succumb easily.

In this paper, we consider a new scenario that describes how the user can be lured to aid the malicious developer. A smartphone includes a screen where application installed by the user displays interfaces. A *screen in an app* is represented by an activity. Apart from activity region, buttons region and border regions are included. Using techniques previously state, when a malicious application is started, malicious graphical interfaces will be presented to the user. Our interest in this work is focused on screen portions that are not explicitly highlighted, but trigger of hidden malicious actions, once the users click or press them inadvertently. We observe at this time, suspect flows transiting inside the system. In fact, the attacker takes advantage again of the device; constituting, therefore, our finding of a new type of attack.

Figure 1 illustrates our methodology. The user intervenes in many stages from the infiltration to the leak of information to the attacker. The goal of the work described in this paper is to analyse the behaviour of the malware in step 3 when the user is doing anything different to what is required by the application. Events we observe are *clicking* or *pressing* on app region different to the usual app interfaces (button, links, input fields, etc.) presented.

In steps 4 and 5, the application can behave as threats of social engineering [9]. Even if the difference here is that the interaction region is not programmed by the malware author but managed by Android system. This can therefore be a flaw of the system.

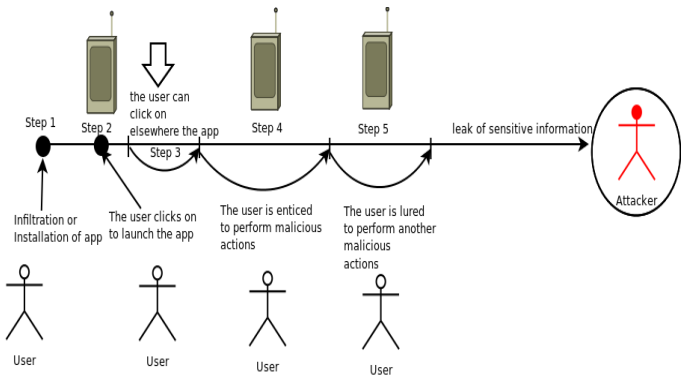

Figure 1: User our methodology

We perform dynamic analysis using *Strace* [26] used to trace system calls and signals on Linux. It allows learning app behaviour effectively through system calls on Android as it is based on Linux. This operation is performed in the virtual environment *Santoku* [13] on Galaxy Nexus with Android 4.2.2. We work on *Droidream* samples from the malwares dataset released for research community [12].

Our contributions are as follows:

- Performing an analysis of system calls: we install a virtual environment of analysis for Android and use tools to understand the behaviour of malware in lower level.
- Description of kernel calls: we describe the entire environment around system calls on Android. This shows the complexity that exists in the process of translating functions written in the application layer to system calls in the Kernel layer.
- To detect a flaw on Android system: We describe a concrete malware analysis scenario and present the details of our implementation. In particular, most of the samples do integrate calls to wait for events for a certain time before continuing the execution. A random event (unwanted) by the user allows performing malicious actions by the malware.

2. COMPONENTS OF ANALYSIS AND METHODS

2.1 Layer of Analysis

The Android platform is built like every other platform: as a stack with various layers running on top of each other, lower-level layers providing services to upper-level services [15].
The application layer: Applications run at this layer, which includes applications written by any third-party developer.
T*he application framework layer:* This layer provides generic functionality to equip developers with a mean for building applications such as APIs[1].

[1] Application Programming Interface

Android application runtime layer: It is used for efficiently running programs on devices with limited resources such as battery, memory, limited, screen display.

The native libraries: Modules of code written in native (such as C/C++) that are compiled down to native machine code for the device and provide some of the common libraries that are available for apps.

The kernel layer: Android runs on top of a Linux 2.6 kernel and interacts with the device hardware providing some core functionalities.

This layer is the one where we perform the analysis of system calls triggered when app starts.

2.2 Kernel Calls

In Android, abstraction in the application layer hides many system details. The complexity of hardware access is hidden to developers and seems simple. They only require in their code to invoke a function or method that deals with the desired action. However, this obfuscation of system details also veils security vulnerabilities of the system such as Zygote vulnerabilities [17].

Most of the time, existing security solutions focus on the application framework layer or upper [18]. Monitoring security for both application framework and native code will be very helpful to mitigate attackers actions. Overseeing applications in low-level layer implies to analyse native code calls in the atomic level called *system calls*.

To access to hardware devices, applications pass through kernel calls. Armando et al. [16] clearly sub-grouped kernel invocations into three types:

System calls: they are used to directly invoke native functionalities of the Kernel.

Binder calls: they support the invocation of the Binder driver at the Linux kernel.

Socket calls: these calls allow read/write data and commands from/to a Linux socket. They do not trigger directly Kernel functionalities in response to commands written on the socket by means of system calls.

In the scope of this paper, we will group all these calls into one called *system calls*. Each system call assigns the control to the Android kernel while a specific function occurs.

2.3 Test Bed

In the rest of this section, click interaction will also refer to press (or touch) interactions. For our work, we use an Android integrated analysis environment called *Santoku* [13]. This is a bootable Linux environment designed to work in Virtual machine. It includes pre-installed platform SDKs, it

integrates the latest security tools and utilities for Android for examining mobile malware.

To directly install applications, we used the Android Virtual Device Galaxy Nexus with Android 4.2.2 – API Level 17. It has ARM CPU and 1 GB RAM, 500 MB internal storage and 200 MB SD card features.

2.4 Samples and Process

We used *DroidDream* samples malwares [12] for analysis. The process is built on the following steps:

- Starting of the Android Virtual Device from the SDK.
- Installation of binaries (*adb install xxx.apk*)
- Use of ADB (Android Debug Bridge) to emulate the device (*adb shell).*
- Identify and retrieve the parent process of the application installed (*ps*).
- Entry point to trace system calls of the process (*strace -p pid*).

Additional tools help us to complete the analysis

- Use of *apktool* [29] to decompile apk resource file into a folder (*java -jar apktool.jar d -d xxx.apk folder*).
- Use of smali/baksmali [27] to diassemble the dex files (*java -jar baksmali classes.dex).*
- Use of *d2j-dex2jar* [28] to decompile .apk to .jar (*sh d2j-dex2jar.sh xxx.apk*)
- Use of JD-UI [30] to visualise from class files the source code Java.

3. RESULTS AND DISCUSSIONS

We first outline the main and suspicious system calls that we extract from our analysis as well as their definition. Due to the lack of completeness of the Android documentation, we referred also to Linux documentation to complete details on Android system calls [19][22].

We define three regions: *inside*, *outside*, *interact, and interest*. The first one (Figures 2 and 3) concerns all the points that are within the main activity and the second one delimits points outside the main activity of the installed application. The *interaction* zone is the one that we did not access in our work. The *interest* zone is the difference between the *interaction* and the *inside* zone.
Our strategy then is to apply click events on points for *interest* and *outside* zones to see responses from the application. The reliability of our results is based on the fact that we do not apply interaction with interface proposed to the user (that is *interaction zone*). In particular, we avoided to click the buttons “agree” and “refuse” reserved surely by the attacker to delude the user in Figure 2 or to click on checkbox in Figure 3.

We have encountered thousands of system calls per application until we stop their execution. It is not feasible to enumerate all but just focus on system calls that generate bad activities in background. Also, we just work on *DroidDream* family so because of the pages limit we just choose two malicious samples *Super History Eraser* and *Task Killer Pro* [25].

*clock_gettime(clockid_t clk_id, struct timespec *tp)*: retrieves the time of the specified clock clk_id.

*int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout)*: blocks for *timeout* milliseconds waiting for an I/O event on a descriptor file.

*read(int fd, void *buf, size_t count)*: attempts to read up to count bytes from file descriptor fd into the buffer starting at buf.

*readv(int fd, const struct iovec *buffer, int count)* & writev(int fd, const struct iovec *buffer, int count):

The first one reads *count* blocks from the file associated with the file descriptor *fd* into the multiple buffers described by *buffer*.

writev(): writes at most count blocks described by *buffer* to the file associated with the file descriptor *fd*.

open(): Given a pathname for a file, returns a file descriptor, which can be written, red, and executed by a process using a permission.

*access(const char *pathname, int mode)*: checks if the caller process has rights to access to file *pathname*.

*recv(int s, void *buf, size_t len, int flags) & recvfrom(int s, void *buf, size_t len, int flags,struct sockaddr *from, socklen_t *fromlen)*: allow to read incoming data from the remote side locally or via network.

*rename(const char *oldpath, const char *newpath)*: renames a file in moving if necessary.

*send(int sockfd, const void *buf, size_t len, int flags)* and *sendto(int sockfd, const void *buf, size_t len, int flags,*

*const struct sockaddr *dest_addr, socklen_t addrlen)*: are used to transmit a message to another socket. The message is found in *buf* and has length *len*.

*mprotect(void *addr, size_t len, int prot)*: changes protection for the calling process memory page(s) containing any part of the address range in the interval [*addr*, *addr*+*len*-1].

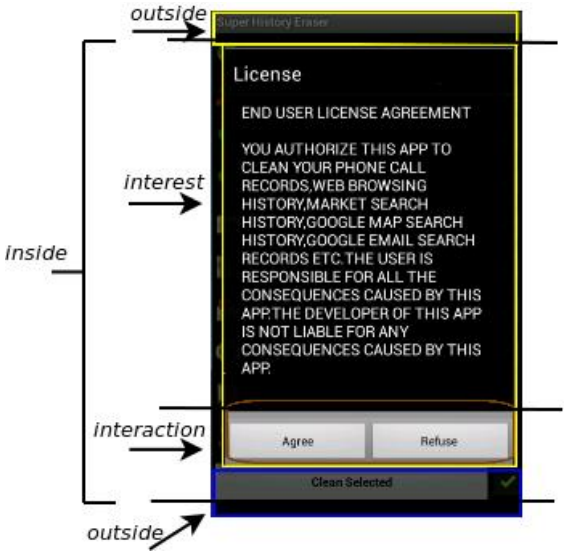

Figure 2: Super History Eraser

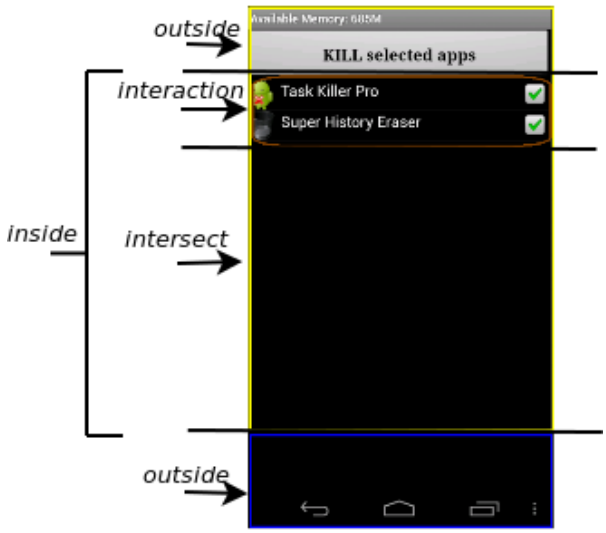

Figure 3: Task Killer Pro

At the beginning, when the user launches the application, it blocks waiting for user to interact by selecting, filling form, validating with buttons. These actions are predefined in the malware by the attacker to communicate with the user. In other words, the application can consider also events different to those provoked by the interface displayed to the user.

Below is a screenshot (Figure 4) with Super History Eraser (after the clicking in *outside*) that shows a flow where a bidirectional communication starts between the main activity and an "unknown" one. As displayed, this is well synchronised using the system call *clock_gettime()* that helps to test if it is the time the launch a fraudulent actions. Then, *epoll_wait()* blocks for events.

```
recvfrom(36, "\2\0\0\0*\213\0\0\0
```

related vicious process was observed before the launching. Moreover, we demonstrate that there exists effectively another *social engineering* [9][25] threat that lures also the user. But with the main characteristic that the attacker takes the hands without receiving a physic interaction response from the user but rather in receiving a click event on smartphone regions seems invulnerable. We assume that this vulnerability is difficult to pick by existing techniques such as LeakMiner [1]. However, this work is limited to just few malwares. It will be helpful to work on all the dataset concluding more Android architectures in order to to analyse further details. Additional tools like Logcat [31] should be used to have more complete results. Even, benign applications should be monitored to see if this applies also to them.

4. LITERATURE REVIEW

Our work on analysis of system calls is not the first. Some studies have been done, even though several used not only system calls to decide but also other features like API calls and permissions. Our particularity is that we scrutinised events generated only by system calls initiated just at the start-up of malwares to detect anomalies.
Monitoring of system calls in Android is discussed by Blasing et al. [2]. They proposed a tool called Android Application Sandbox (AASandbox) that performs both static and dynamic analysis. It disposes a module which monitors system calls that logs the return value of each system call independently to the parameters. Our work brings a completion to their work in specifying a type of social engineering threats evolving from analysis of signatures, parameters and return value of calls.

Armando et al. [3] built a kernel module that logs system calls invoked by Application Framework layer and a tester application capable to read the logs and re-execute successfully the tracked calls. This module re-executes the tracked calls and this information collected after analysis indicate that little control is exercised among the Android and the Linux layers, thereby indicating that the attack surface of the Android platform is wider than expected. Our results come to confirm this result.

Wu et al. proposed DroidMat [4] that detects malwares through the Manifest file and traces of API calls. They demonstrated that this tool can find more Android malware than Androguard. However, DroidMat is limited to detect the Android malware because malware families that are found, are only with a single sample and therefore it is not easy to learn their behaviour. Moreover, there are two families (BaseBridge and DroidKungFu) of malwares that cannot well be detected by DroidMat.

In the same direction, Isohara et al. [5] proposed in 2011 a kernel-based behaviour analysis for android malware inspection. This system records system calls and filters events with the application. They use policies based on signatures of information leakage to detect a malicious activity. With this approach, they showed that their system can effectively detect malicious actions of the unknown applications. Some of these actions that we found in our work can be to lure user to click somewhere on the screen.

In 2012, Zheng et al. [24] proposed SmartDroid, a static analysis to extract expected activity switch paths by analysing both Activity and Function Call Graphs, and then uses dynamic analysis to traverse each UI elements and explore the UI interaction paths towards the sensitive APIs. They prove that based on this graph they can detect UI-based trigger conditions to expose sensitive behaviour of several Android malwares not detected by Taindroid.

5. CONCLUSION

We show a new way of luring the Android users. In the meantime, it is not a UI programmed by the malware author. It is the phone interface. Actions on it can be malicious even without the knowledge of the attacker rather in his advantage. We realised a scenario to monitor System calls initiated by DreamDroid samples when the user clicks to launch the malware application. We discovered that, the click event participate to the malicious programmer tasks. We observed that even if the attacker clicks outside the application interface malicious actions can be triggered. Our experimental analysis demonstrates that Android framework must consider this pitfall.

As future work, we will extend our research to many Android malware families and other Android versions to generalise the behaviour. A classification will help us to take decisions against bad actions. We will develop a mechanism to mitigate this problem and to inform users on the possible malicious actions.

REFERENCES

[1]. Zhemin, Y. and Yang, M. LeakMiner: Detect Information Leakage on Android with Static Taint Analysis. In 2012 Third World Congress on Software Engineering (WCSE), 101–104, 2012., doi:10.1109/WCSE.2012.26.

[2]. Blasing, T., Batyuk, L., Schmidt, A.-D., Camtepe, S. and S. Albayrak. An android application sandbox system for suspicious software detection. In Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on, pages 55-62., 2010.

[3]. Armando, A., Merlo, A. and L. Verderame. Security Issues in the Android Cross-Layer Architecture. 2012

[4]. Dong-Jie, W., Mao, C., Wei, T., Lee, H. and W. Kuo-Ping. DroidMat: Android Malware Detection through Manifest and API Calls Tracing. In 2012 Seventh Asia Joint Conference on Information Security (Asia JCIS), 62–69, 2012, doi:10.1109/AsiaJCIS.2012.18.

[5]. Isohara, T., Takemori, K. and A. Kubota. Kernel-based Behavior Analysis for Android Malware Detection. In 2011 Seventh International Conference on Computational Intelligence and Security (CIS), 1011–1015, 2011., doi:10.1109/CIS.2011.226.

[6]. Gartner Says Worldwide Sales of Mobile Phones Declined 2.3 Percent in Second Quarter of 2012. http://www.gartner.com/newsroom/id/2120015, accessed on 09.06.2013

[7]. Global market share of smartphone operating systems by quarter 2009-2012 http://www.statista.com/statistics/73662/quarterly-worldwide-smartphone-market-share-by-operating-system-since-2009/, accessed on 09.06.2013

[8]. Malicious Mobile Threats Report 2010/2011,http://juniper.mwnewsroom.com/manual-releases/2011/At-Risk--Global-Mobile-Threat-Study-Finds-Security, accessed on 09.06.2013

[9]. Lookout Mobile Threat Report August 2011. www.mylookout.com/_downloads/lookout-mobile-threat-report-2011.pdf, accessed on 09.06.2013

[10]. Lookout Mobile Security Technical Tear Down.https://blog.lookout.com/wp-content/uploads/2011/03/COMPLETE-DroidDream-Technical-Tear-Down_Lookout-Mobile-Security.pdf, accessed on 09.06.2013

[11]. Castillo, C. Android Malware. Past, Present, and Future. McAfee Inc, 2011.

[12]. Malware dataset for research community http://malgenomeproject.org/, accessed on 09.06.2013

[13]. Santoku https://santoku-linux.com/, accessed on 09.06.2013

[14]. Chasing Android System Calls Down The Rabbit Hole.http://cgi.cs.indiana.edu/~nhusted/dokuwiki/doku.php?id=research:downtheandroidhole1, accessed on 09.06.2013.

[15]. Six, J.Application Security for the Android Platform: Processes, Permissions, and other Safeguards. O'Reilly, Sebastopol, CA 2011

[16]. Armando, A., Merlo, A. and Verderame, L. An Empirical Evaluation of the Android Security Framework, in Proc. of the 28th IFIP TC 11 International Information Security and Privacy Conference (SEC 2013), 8-10 July 2013, Auckland (New Zealand).

[17]. Armando, A., Merlo A., Migliardi, M. and Verderame, L. Would you mind forking this process? A denial of service attack on Android (and some countermeasures). In Proc. of the 27th IFIP International Information Security and Privacy Conference (SEC 2012), IFIP Advances in Information and Communication Technology, 376, pages 13-24. Springer, 2012

[18]. Enck, W.Defending Users against Smartphones Apps: Techniques and Future Directions, 2011

[19]. Android system calls. https://github.com/android/platform_bionic/blob/master/libc/SYSCALLS.TXT, accessed on 09.06.2013.

[20]. Brady, P. Anatomy and physiology of an Android. Google I/O, 2008

[21]. Schreiber, T. Android Binder, Android Interprocess Communication, 2011

[22]. Linus documentation. http://linux-documentation.com/en/man/man2/, accessed on 09.06.2013

[23]. Zhou, Y. and X. Jiang. Dissecting Android Malware: Characterization and Evolution. 2012 IEEE Symposium on Security and Privacy, 2012

[24]. Cong, Z., Zhu, S., Dai, S., Gu, G., Gong, X., Han, X. and Zou, W. SmartDroid: An Automatic System for Revealing UI-based Trigger Conditions in Android Applications. In Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, 93-104. SPSM '12. New York, NY, USA: ACM, 2012., doi:10.1145/2381934.2381950.

[25]. Zhou, Y. and Jiang, X., "Dissecting Android Malware: Characterization and Evolution," Security and Privacy (SP), 2012 IEEE Symposium on, vol., no., pp.95, 109, 20-23 May 2012

[26]. Strace Linux man page http://linux.die.net/man/1/strace, accessed on 09.06.2013

[27]. Smali, http://code.google.com/p/smali/, accessed on 07.06.2013

[28]. Dex2jar, http://code.google.com/p/dex2jar/, accessed on 07.06.2013

[29]. Android-apktool, a tool for reverse engineering Android apk files, http://code.google.com/p/android-apktool/, accessed on 07.0621013

[30]. JD-GUI, http://java.decompiler.free.fr/?q=jdgui, accessed on 07.06.2013

[31]. Logcat,http://developer.android.com/tools/help/logcat.html, accessed on 07.06.2013