Detecting SMS-based Control Commands in a Botnet from Infected Android Devices

Anh Nguyen
School of Information Technology
Deakin University
Burwood, VIC 3125, Australia
Email: anguyen@deakin.edu.au

Lei Pan
School of Information Technology
Deakin University
Burwood, VIC 3125, Australia
Email: l.pan@deakin.edu.au

Abstract

An increasing number of Android devices are being infected and at risk of becoming part of a botnet. Among all types of botnets, control and command based botnets are very popular. In this paper, we introduce an effective and efficient method to detect SMS-based control commands from infected Android devices. Specifically, we rely on the important radio activities recorded in Android log files. These radio activities are currently overlooked by researchers. We show the effectiveness of our method by using the examples from published literature. Our method requires much less user knowledge but is more generic than traditional approaches.

Index Terms

botnet; logging; Android; digital forensics

1. Introduction

With the increasing number of botnets and widespread of malware, a large number of Android devices are being infected and thus become a part of botnet. Android devices are attractive and vulnerable, due to their design features as a free and open-source operating system for mobile devices [1]. Furthermore, Android systems have a weak security model in which special permissions are granted easily to perform malicious tasks.

The interest of building a botnet with mobile devices is shown fairly recent. Many mobile devices become part of command and control botnets [2]. Zeng et al. [3] build a simple botnet with less than 200 control commands on a mobile network. They also show that even if an innocent user deletes a control message sent via SMS, the command has been regardlessly executed upon receipt of the SMS.

Traditional approaches to unveil the existence of a bot take a long time. Our research question is to effectively and efficiently detect the control commands from an Android device transmitted within a botnet. We take advantage of Android log files which are embedded in the kernel level, and in particular, the radio activity logs which record many important system events related to botnet activities. Our contributions in this paper include a detailed introduction of the Android log system and the identification of three important radio activities associated with SMS-based botnet activities. Our new detection method requires much less user knowledge but is more generic than traditional reverse engineering approach to decompile the binary files into source codes.

The rest of this paper is organized as follows: Section 2 lists the related work on Android malware analysis; Section 3 illustrates Android log files; Section 4 describes our detection method, our experiments and the results; and we conclude in Section 5.

2. Literature Review

There are three commonly used malware analysis approaches for Android devices — reviewing source codes, applying a sandbox and analyzing Android kernel information. Reviewing the source codes of a program is a reliable approach for analyzing Android malware. For example, Blasco [4] analyzes an Android malware package (named `RU.apk`) by reverse engineering the binary file into source codes. The *apk* package is firstly uncompressed by using a ZIP program; then Blasco locates the `class.dex` file from the uncompressed files; a Google tool named *Dex2jar* is then applied to convert `class.dex` to `class.jar`; finally, the Java decompiler *jd* is applied to retrieve source codes from `class.jar`. More specifically, the app `RU.apk` secretly forwards all incoming SMS messages to two Russian premium numbers. Blasco's case study shows the power of reverse engineering an app package *apk*; however, it is not applicable when the *apk* file is unavailable. The second approach to analyzing a botnet is to use a sandbox. A sandbox is injected to an Android device where any use of special permissions such as calling a phone, sending SMS, connecting to the Internet is handled by the sandbox instead of the handset [5], [6]. The third approach is to analyze kernel-level information. As revealed in [7], [8], different apps revoke system calls differently according to the recorded information in kernel-level log files. The collected log files contain the filtered events with all system calls from the target application. By using this method, Isohara et al. build app signatures based on the intensity of I/O and process management operations.

However, none of the above approaches applies to our case. Our reasons are threefold: It is generally infeasible to acquire botnet source codes for many botnets do not use a static binary program or an app package; the sandboxing technique is complicated and often blocks bot's activities; botnets evolve so quickly that fixed signatures rarely work for a long period of time. Inspired by all of the existing work we focus on the recorded behaviors of a handset.

Lessard and Kessler [9] list a few forensic methods to retrieve information from an Android device. One of the practical combinations of techniques is to firstly enable USB debugging, then to gain the root access to the device, and lastly to create *dd* disk images of the mounted file systems on Android. These acquired image files can be analyzed with popular forensic tools such as *Forensic ToolKit (FTK)* and *EnCase*. However, this process is labor-intensive and error-prone. Zhong et al. [10] use a customized Google map on Android device to retrieve an individual user's geo-spatial data which is then sent to the police via 3G network. Though sending data to the police seems to be safe and reliable, it brings excessive concerns of privacy and misuse of personal data. An alternative approach is to use commercially available and automated phone forensic tools such as *Device Seizure* from Paraben and *Oxygen forensic suite*. Both approaches require the USB debugging mode on the handset to be enabled. However, changing the setting of a handset may alter digital evidence stored onboard. And it is challenging to enable any write blocking mechanism on a handset.

The next section will introduce Android logging system and log files.

3. Android Log Files

Google modified the original Linux kernel to suit Android. More than 160 kernel files are created or modified to support new 'Log Device' and 'Android Debug Bridge' (*adb*). Hence, Android systems use *syslog* for the unmodified Linux kernel and its own app log for the new 'Log Device' and 'Android Debug Bridge'. The *syslog* file format is defined as an international standard in RFC5424 [11], and the app log file format is defined by Google.

3.1. Android Syslog

Syslog has a hierarchical structure and a simple syntax as follows:

- A log file consists of multiple *syslog* messages.
- Each *syslog* message has two or three parts: the first part is a header; the second part is structured data; and the third part is an optional message. A white space symbol is inserted between each part to separate them.
- The header consists of a priority value, the syslog version, a timestamp, a hostname, an application name, a process ID and a message ID.
- The structured data is a list of parameter names and their associated values.
- The optional message contains the detailed message which is stored in ASCII or in UTF-8 codes.
- By default, a missing value is indicated by a hyphen symbol in ASCII.

Comparing with a standard Linux system, Android system excludes the program `syslogd`. So, the ordinary Linux log directory `/var/log` does not exist on Android; instead, an equivalent file `proc/kmesg` is created on Android for logging Linux kernel messages. There are two methods to access this file — either by using the Linux command `dmesg` through the 'Android Debug Bridge' (*adb*) or by directly accessing the file via the file system.

Originally designed for hardware debugging purposes, kernel messages mainly contain basic information of the hardware and the operating system. Since these kernel messages keep track of the boot process, the running system processes and their process IDs. These messages have forensic values. Below is an example of kernel message from an Android device. Specifically, the following log entries include some warning, informational and notice messages generated during the boot process: The kernel information with a timestamp is the first message; then the next three messages show the

Android device has a ARMv7 processor and 512MB RAM. The fifth message shows the memory mapping; and the rest indicate the process of mounting three storage devices `mtdblock1`, `mtdblock2` and `mtdblock3` as `yaffs` file system.

```
<5>Linux version 2.6.29-gc497e41       \
(kroot@kennyroot.mtv.corp.google.com) \
(gcc version 4.4.3 (GCC) ) #2 Thu Dec \
8 15:07:43 PST 2011
<4>CPU: ARMv7 Processor [410fc080]     \
revision 0 (ARMv7), cr=10c5387f
<4>CPU: VIPT nonaliasing data cache,   \
VIPT nonaliasing instruction cache
<6>Memory: 512MB = 512MB total
<5>Memory: 515712KB available (2756K  \
code, 683K data, 108K init)
<6>yaffs: dev is 32505856 name is      \
"mtdblock0"
<4>yaffs: Attempting MTD mount on 31.0\
, "mtdblock0"
<6>yaffs: dev is 32505857 name is      \
"mtdblock1"
<4>yaffs: Attempting MTD mount on 31.1\
, "mtdblock1"
<6>yaffs: dev is 32505858 name is      \
"mtdblock2"
<4>yaffs: Attempting MTD mount on 31.2\
, "mtdblock2"
```

The numbers appearing in the beginning of each message are bounded by brackets. According to [11], each of these numbers indicates priority values. A priority value is calculated by multiplying the facility number by 8 and adding the severity level. And kernel messages have zero as their facility number. Since Android only stores kernel messages in *syslog* format, the priority values in the above example are equal to the default *syslog* severity levels:

0 Emergency: system is unusable
1 Alert: action must be taken immediately
2 Critical: critical conditions
3 Error: error conditions
4 Warning: warning conditions
5 Notice: normal but significant condition
6 Informational: informational messages
7 Debug: debug-level messages

3.2. Android App Log

Android app logs are handled by the Android Logging system, which is completely separate from the Linux kernel log system. Unlike hardware information, the Android Logging System emphasizes the status of the running programs on the device. It includes four basic components: An Android kernel driver called `logger`, four logging buffers allocated to store log messages — *events*, *main*, *radio* and *system*, a standalone program `logcat` to view log messages, and a programming interface (via *Eclipse* and *DDMS*) designed to view and filter the logs.

To illustrate the Android Logging System, we use the picture from [12] in Figure 1. There are two devices in the figure — a target and a host. The target is the Android device whose logs are being analyzed, and the host is the computer to which the device is connected. On the target, the log files are stored in the `/dev/log` directory at the lowest level. Corresponding to the four log buffers, the

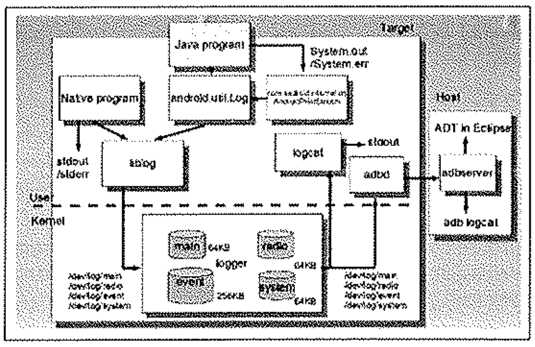

Figure 1: Android Logging System from [12]

log files are `/dev/log/main`, `/dev/log/radio`, `/dev/log/event` and `/dev/log/system`. In `/dev/log/radio`, we often find the log messages with message tags "RIL", "AT", "RILD", "RILC", "RILJ", and "GSM". All of the log files are 64 KB except that `/dev/log/event` is 256 KB. On each Android handset, the log sizes are fixed so that the newer records will replace the earlier ones. The contents of each record are generated with the `liblog` library before they are stored in these log files. As shown in Figure 1, the `liblog` library is in charge of recording all application logs. The native Android apps can directly access `liblog`, and the third-party apps should use the API `android.util.log` to access `liblog`. In addition to outputting the log messages to the log files, the programs may display the messages on the device screen via `stdout` and `stderr` channels.

Similar to the kernel log messages, each Android log message contains multiple parts — a message tag indicating the application that the message originated from, a timestamp, the priority of the event represented by the message and the log message itself. The `logcat` tool chronologically sorts the events and can be invoked on the target or on the host through the Android Debug Bridge Daemon (*adbd*). By default, the Android `logcat` tool only outputs three parts of the log message — the priority tag denoted by one of the 5 letters ('E' for ERROR, 'I' for INFO, 'D' for DEBUG, 'W' for WARN, and 'V' for VERBOSE), the message tag indicating the message origin and the log message itself. Furthermore, the separation symbols vary between different parts: a slash symbol '/' is immediately after a priority tag and before a message tag, and a colon symbol ':' is immediately after the message tag and before the log message.

The following examples are the outputs from `logcat`. The command `logcat -b main` retrieves the main application log `/dev/log/main`, and the following error message indicates the absence of an error trace file:

```
E/Trace    (264): error opening trace \
file: No such file or directory (2)
```

The command `logcat -b events` retrieves the system events log `/dev/log/events`, and the following information message indicates the start of Android bootup process:

```
I/boot_progress_start(36): 22060
```

`logcat -b radio` retrieves the radio and phone-related information log `/dev/log/radio`, and the following debugging message indicates an incoming phone call:

```
D/AT      (  943): AT< RING
```

The command `logcat -b system` retrieves the system debugging log `/dev/log/system`, and the following error message indicates a run-time error caused by missing a method:

```
E/AndroidRuntime( 8681): java.lang.NoSuch\
MethodError: com.herongyang.AboutAndroid.\
getObbDir
```

The next section shows our proposed method of using Android log files to detect botnet control commands via SMS.

4. Using Android Logs to Detect Botnet Control Commands

Our detection method is to search for radio events. Our reasons are 1) botnet uses radio as a channel to send and receive short control commands to meet the need of frequent communications to its zombies; 2) the radio log on actual handsets is large enough to store generally a few days' telecommunication logs; 3) the address of destination nodes and source nodes are stored in clear text in the radio logs; 4) the radio logs are well kept in the kernel so that a complete erase of all entries is difficult.

Specifically, we focus on two categories in `/dev/log/radio`: The first category is to retrieve the device information such as `D/RIL onRequest: GET_IMSI` and `D/RIL onRequest: GET_IMEI` which retrieve the device IMSI and IMEI numbers respectively; the second category is to send SMS messages such as `D/RIL onRequest: SEND_SMS`. Our detection method is very simple but effective. We show two examples below.

To carry out the experiment, we use the information of a prepaid Optus device and the same SMS examples from [3]. The first example short message reads as:

> Free ringtones download at www.myringtone.com, using username VIP, password YTJiNGQxMWw to log on

The only obscured text is "YTJiNGQxMWw" which is a Base-64 encoded string "a2b4d11l". Without knowing what this number means, we acquire the radio log of the recipient's device. By filtering out the irrelevant entries and the ones before the recipient of the SMS, we obtain the following 3 entries:

```
D/RIL (  32): onRequest: GET_IMSI
D/RIL (  32): onRequest: GET_IMEI
D/RIL (  32): onRequest: SEND_SMS
```

Now, the behavior of this apparent spam SMS triggers our device to retrieve system information (at least IMSI and IMEI numbers) and then send a new SMS to somewhere else. We then examine the radio log and find that our IMSI and IMEI numbers have been sent to an American number +14083631980. The following debugging message is located in one of the later events:

```
D/AT   (  32): AT> AT+CMGS="+14083631980"\
,145 IMSI=085950200100263914            \
IMEI=353502020487979^Z
```

where the command "AT+CMGS" indicates that the SMS has been sent to +14083631980, "145" is the international phone number format, and the characters before "^Z" are the SMS body. Our findings are consistent to the original paper, which the first SMS is a bot command "SEND_SYSINFO a2b4d11l".

The second SMS is also from [3]:

> Your paypal account was hijacked (Err msg: NzkxMjAzNDIxODExMDUyM183Mz). Respond to http://www.bhocxx.paypal.com using code Q3MDk2NDUyXzEyMzQ1Njc4

This time, we need to concatenate the two encoded strings "NzkxMjAzNDIxODExMDUyM183Mz" and "Q3MDk2NDUyXzEyMzQ1Njc4". We use Base-64 to decode the result string as "791203421811O523_7347096452_12345678". Without knowing what these three numbers mean, we acquire the radio log of the recipient's device. By filtering out the irrelevant entries and the ones before the recipient of the SMS, we obtain the following entry:

```
D/RIL (  32): onRequest: SEND_SMS
```

Now, this apparent spam SMS lets the device send out a SMS. We examine the radio log and find that a blank SMS has been sent to an American number +17347096452. The following debugging message is located in one of the later events:

```
D/AT   (  32): AT> AT+CMGS="7347096452",\
129 ^Z
```

where the command "AT+CMGS" indicates that the SMS has been sent to 7347096452, "145" is the phone number format without the country code, and the characters before "^Z" indicate an SMS with an empty body. Our findings are also consistent to the original paper, which the second SMS is a bot command "FIND_NODE 7912034218110523_7347096452_12345678". According to [3], this command attempts to locate a bot whose ID is 7912034218110523 with passcode 12345678 to the bot whose phone number is 7347096452.

In this section, we propose our detection method and apply it to treat the two disguised SMS messages. We successfully detect botnet control commands with the help of the log file `/dev/log/radio`. The success of our method relies on the integrity and trustworthiness of the radio log file which is currently overlooked by many researchers. As a simple solution to the log integrity problem, we have implemented an Android program which captures all the log entries in real-time and stores the log entries in a SQLite database on external storage devices. We believe that this mitigation method would help to prevent the loss of Android logs. We conclude this paper in the next section.

5. Conclusions and Future Work

In this paper, we explain the Android logging system in details. We identify a few indicative yet significant logging symbols to effectively detect the control commands of a botnet. Our method relies on the Android radio log which is overlooked by most researchers. We show the effectiveness of our method by using the examples from published literature. Our method requires much less user knowledge but is more generic than traditional reverse engineering approach.

Our future work is to improve the current method so that it does not require a non-tampered radio log file. Since validating log entries is beyond the scope of this paper, our next step is to distinguish whether a log message is genuine and to recover the deleted or altered log messages prior to the log analysis.

References

[1] R. Meier, *Professional Android 2 Application Development.* John Wiley & Sons, 2010.

[2] C. Xiang, F. Binxing, Y. Lihua, L. Xiaoyi, and Z. Tianning, "Andbot: towards advanced mobile botnets," in *Proceedings of the 4th USENIX conference on Large-scale exploits and emergent threats*, ser. LEET'11, 2011, pp. 11–17.

[3] Y. Zeng, K. G. Shin, and X. Hu, "Design of sms commanded-and-controlled and p2p-structured mobile botnets," in *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, ser. WISEC '12, 2012, pp. 137–148.

[4] J. Blasco, "Introduction to android malware analysis," *INSECURE Magazine*, vol. 34, pp. 25–37, 2012.

[5] T. Blasing, L. Batyuk, A.-D. Schmidt, S. Camtepe, and S. Albayrak, "An android application sandbox system for suspicious software detection," in *Proceedings of the 5th International Conference on Malicious and Unwanted Software (MALWARE)*, oct. 2010, pp. 55–62.

[6] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen, "I-arm-droid: A rewriting framework for in-app reference monitors for android applications," in *Proceedings of IEEE Mobile Security Technologies (MoST)*, 2012.

[7] T. Isohara, K. Takemori, and A. Kubota, "Kernel-based behavior analysis for android malware detection," in *Proceedings of the Seventh International Conference on Computational Intelligence and Security (CIS)*, dec. 2011, pp. 1011–1015.

[8] P.-M. Chen, H.-Y. Wu, C.-Y. Hsu, W.-H. Liao, and T.-Y. Li, "Logging and analyzing mobile user behaviors," in *Proceedings of International Symposium on Cyber Behavior*, 2012.

[9] J. Lessard and G. C. Kessler, "Android forensics: Simplifying cell phone examinations," *Small Scale Digital Device Forensics Journal*, vol. 4, no. 1, pp. 1–12, 2010.

[10] B. Zhong, L. Niu, and H. Chen, "Design and implement of 3g mobile police system," in *Proceedings of the 2nd International Conference onConsumer Electronics, Communications and Networks (CECNet)*, april 2012, pp. 1752–1755.

[11] R. Gerhards, "The Syslog Protocol," RFC 5424 (Proposed Standard), Internet Engineering Task Force, March 2009. [Online]. Available: http://www.ietf.org/rfc/rfc5424.txt

[12] T. Kobayashi, "Logging system of android," 2010, presented at CELF's Japan Technical Jamboree 34.