Possible Keyloggers Without Implementing a Keyboard in Android

Itzael Jiménez Aranda(✉), Eleazar Aguirre Anaya, Raúl Acosta Bermejo, and Ponciano Jorge Escamilla Ambrosio

Instituto Politécnico Nacional - Centro de Investigación en Computación, GAM, 07738 Mexico City, Mexico
itzaelja@gmail.com, eaguirrea@ipn.mx, {racosta,pescamilla}@cic.ipn.mx

Abstract. Like the main input way to introduce information in the majority mobile devices nowadays is the screen, it is the main source where a malware could get private information. A keylogger, in this way could obtain private information. Researches of this type of malware until this moment are focused on the Android architecture application layer, leaving aside the other layers, so a keylogger could also be implemented in another layer and only use the application layer like the insertion method. An analysis of the data flow when a key is pressed on the screen is presented, from the system call by an interruption caused by hardware, the methods involved in this flow and possible generated logs and related files, performing an experimentation procedure to extract information about the keys pressed in order to determine which points can be used to get private information without the necessity of implement a third-party keyboard.

Keywords: Keylogger · Touchlogger · Malware · Android keylogger Touchscreen

1 Introduction

Nowadays the first option to input information to some mobile device is the screen of the device, so it is the first alternative in order to get private information. Android allows install third-party keyboards, this being something that can be harmful to the user since it can compromise user privacy [1–3]. There are many third-party keyboards that seem to be a simply keyboard with a better design or with extra features, but these could be extracting all information entered from the screen by the user [1].

A keylogger is a software able to record the keys pressed in one system. In mobile devices the key pressed is a virtual key, for that some authors call the keyloggers for the mobile devices as touchloggers. In some cases the keyloggers are used as a legitimate personal or professional IT monitoring tool, but in many cases are used to capture sensitive information, like passwords or financial information, which is then sent to third parties for criminal exploitation [4].

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2018
X. Lin et al. (Eds.): SecureComm 2017 Workshops, LNICST 239, pp. 104–112, 2018.
https://doi.org/10.1007/978-3-319-78816-6_8

The research of the keyloggers in the desktop systems is wide but in the mobile systems is the opposite. So that in the Android system it still continues without has a deep exploration. In general the researches studied how a keylogger is implemented in Android, but in all the cases is about a third-party keyboard installed in the system leaving to one side that can be possible that a keylogger can be implemented in some of the other three layers architecture of Android and just use the application layer as the insertion way to the system, an example of insertion can be a Trojan that of course it not be a keyboard. These researches make a Play Store Keyboards analysis for determine possibles keyloggers, also show the permissions requested by the Android keyloggers and the facility to store the keys pressed in some file and then send the file to some server. As well the researches give recommendations in order to make people aware in how they can avoid this kind of mobiles devices threats [1–3,5].

This work presents an Android keyboard data flow analysis with an experimentation procedure for determine which points can be used by the malware developer for implement a keylogger without the necessity of implement a third-party keyboard.

This article is organized as follow: Sect. 2 describes the research work related to the study of a keylogger on Android. Section 3 describes the problem statement. Section 4 describes the analysis done and the results obtained. And finally the Sect. 5 are conclusions and future work.

2 Related Works

As we mention in one paragraph above, much remains to be studied to the Keyloggers in the field of mobile operating systems. In [1–3] carry out a study of keyboards available in the Play Store to determine the number of possible keyloggers, the amount of requested permissions and permission types are taken into account, so these analysis are performed on the Android architecture application layer. About 80% requests Internet permission and writing memory permission. In [2] perform certain questions to mobile application developers. Why ask Internet permission to develop a third-party keyboard, was one of them and the answer was that may be necessary updates, so not because a third-party keyboard asks Internet permission means to be a keylogger, although with a possible potential to be. In [1] a study is done with the Wireshark traffic tool in the network to determine which keyboards requesting Internet permission are actually extracting information, the result was that 7.9% of the applications analyzed (11 of 139) caused network traffic when an email and password were written, although it is mentioned that the other applications could be time bombs and therefore at that time these did not show network traffic.

In [1,2,5] is demonstrated the ease of obtain information through installing a malicious third-party keyboard malicious, storing and sending information to an external server, building a keylogger, to study what types of permits require, what methods at application level are needed to develop the keyboard and so on. All of these researches are focused on the application layer of the android architecture, but can be a possibility that a keylogger is being implemented in some of the others architecture layers.

In [6] make a record of all the mobile device touch logs with a software for realice a characterization between the device and the user, they use the file */dev/input/eventX* as the way to get the logs.

In [7] studies the iOS data flow for a benign touchlogger and malign touchlogger, making a private and public framework hooking related with the iOS keyboard. The benign part is for to know if the mobile device is been using by the owner. However about the malign part is for get private information, with their method they can get the event type and the touch coordinates, so when a specific app is open the tool register the touches for relate them with the keys pressed known the coordinates and the position of the mobile device.

3 Problem Statement

As any input device, the response to an interruption made from the hardware has a flow, which passes through different stages to reach to the application that corresponds the request, so at some stage might exist some vulnerability that can be exploited if it exist, and use it in order to get private user information.

Therefore, a keylogger can be implemented not as a third-party keyboard, that is not directly in the Android application layer, since it could obtain information through some vulnerable point in the flow of data when any virtual key is pressed.

We make an analysis of the processes and files related with the touchscreen studying the touchscreen data flow when a user press the screen until the information reaches the application performing an exploratory experimentation methodology to extract information about the keys pressed in order to determine which points can be used by the malware developer for implement a keylogger without the necessity of implement a third-party keyboard, so as to get private information. This information can be useful for characterization and a detection mechanism.

4 Touch Screen's Data Flow

In [8] mention a summary of the processes in the Android touchscreen, the Fig. 1 shows the data flow. First "EventHub" reads the raw events from the "evdev" driver. After "InputReader" consumes raw events and updates internal processes statements about the position and other characteristics of each tool. Also it maintains the states of the buttons. If a physical or virtual key is pressed, "InputReader" notifies to "InputDispatcher", also "InputReader" determines whether the touch was made within the limits of the screen and if necessary it notifies to "InputDispatcher". "InputDispatcher" uses to WindowsManagerPolicy, to determine whether the event should be attended. Then "InputDispatcher" releases the event to the appropriate application which is in the application layer.

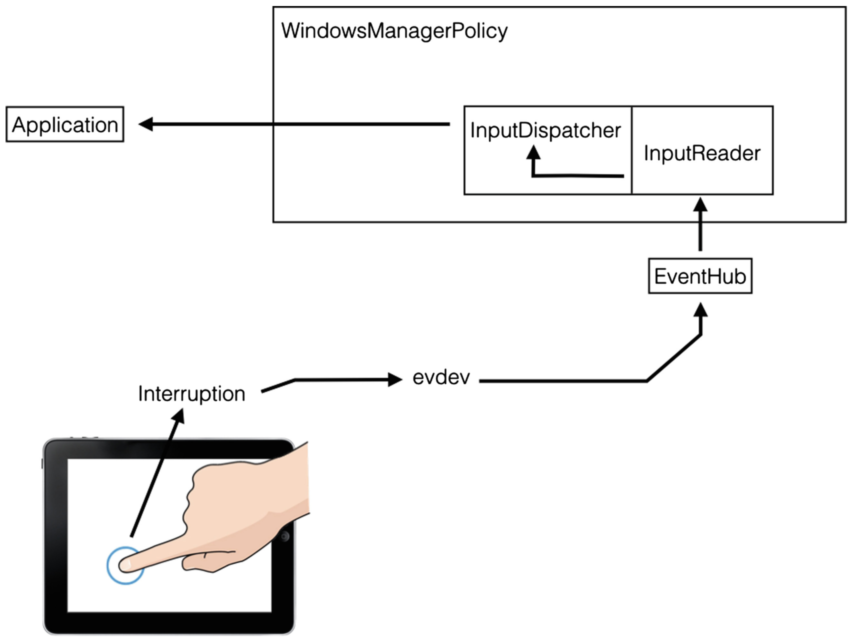


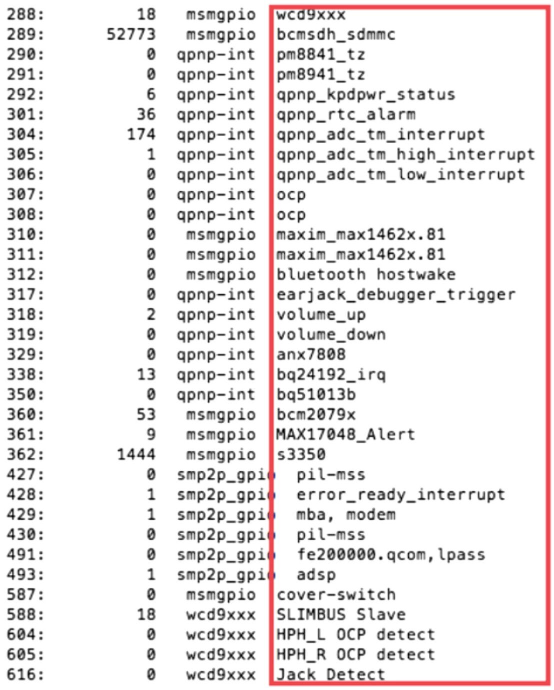
Fig. 1. Data flow when the touch screen is pressed.

For search points where is possible get data about the touchscreen flow is necessary explore the system. Considering the exploratory part is required have full access to the system, and have a native system without unnecessary modifications. For the above the device used is a LG - Nexus 5.

As the driver is the first point where the data of the touchscreen flow pass, this is the start for the exploratory methodology, for analyze its source code and experiment with it and determine if here a malware developer can get information about the keys pressed. As the Android system can be in different brand devices with different I/O devices the Android kernel have different drivers files corresponding to different I/O devices, so is mandatory know the driver which the system is using. Due the driver is loaded like a module to the kernel and it register one function using *request_irq()*, to be able to notify when the user make an interact with the hardware and handle the originated interrupts and the interruption, normally it carry the name of the module loaded in the kernel.

The */proc/interrupts* file provides information about the interrupts functions names of the system as well as the drivers interruptions functions. The Fig. 2 shows a part of how the file provides the information, highlighting the interrupts names, is necessary determine which interruptions correspond to the touchscreen as the names are not clear in a first view.

With the */system/build.prop* file and the Google Git webpage [9] is possible determine and find the system kernel and then the driver source code. So as to determine the driver used by the system we compare the name of each driver in



```
288:        18   msmgpio  wcd9xxx
289:     52773   msmgpio  bcmsdh_sdmmc
290:         0 qpnp-int   pm8841_tz
291:         0 qpnp-int   pm8941_tz
292:         6 qpnp-int   qpnp_kpdpwr_status
301:        36 qpnp-int   qpnp_rtc_alarm
304:       174 qpnp-int   qpnp_adc_tm_interrupt
305:         1 qpnp-int   qpnp_adc_tm_high_interrupt
306:         0 qpnp-int   qpnp_adc_tm_low_interrupt
307:         0 qpnp-int   ocp
308:         0 qpnp-int   ocp
310:         0   msmgpio  maxim_max1462x.81
311:         0   msmgpio  maxim_max1462x.81
312:         0   msmgpio  bluetooth hostwake
317:         0 qpnp-int   earjack_debugger_trigger
318:         2 qpnp-int   volume_up
319:         0 qpnp-int   volume_down
329:         0 qpnp-int   anx7808
338:        13 qpnp-int   bq24192_irq
350:         0 qpnp-int   bq51013b
360:        53   msmgpio  bcm2079x
361:         9   msmgpio  MAX17048_Alert
362:      1444   msmgpio  s3350
427:         0 smp2p_gpi    pil-mss
428:         1 smp2p_gpi    error_ready_interrupt
429:         1 smp2p_gpi    mba, modem
430:         0 smp2p_gpi    pil-mss
491:         0 smp2p_gpi    fe200000.qcom,lpass
493:         1 smp2p_gpi    adsp
587:         0   msmgpio  cover-switch
588:        18   wcd9xxx  SLIMBUS Slave
604:         0   wcd9xxx  HPH_L OCP detect
605:         0   wcd9xxx  HPH_R OCP detect
616:         0   wcd9xxx  Jack Detect
```

Fig. 2. Some system interrupts, marked in red their names. (Color figure online)

the specific kernel in the Google Git with the name of each interruption, but in our case it is not possible perform a relation because neither interrupts names match with the name of the drivers names in the Google Git. So it was not possible decide which driver is used by our system and explore its source code in order to find if a malware could be getting information about the keys pressed. As the driver could not be determined, the decision taken is to explore the first contacts in the user space with the touchscreen. The volatile memory is the first contact that has the process of the keyboard and it is one of the elements in the user space that interact with the touchscreen, so is made an exploration of the volatile memory because possibly it store the keys pressed. Using the adb tool information about the processes executed in the system is got. When the keyboard is being executed, is showed which process ID (PID) correspond to it and its package name, in our case is com.google.android.inputmethod.latin. Knowing the PID we can search the directory and files and analyze them.

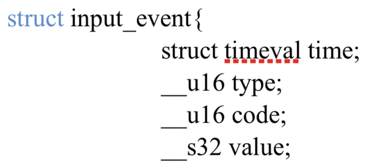
At the process directory there are several files related with the executed process, maps file provides information about the memory section assigned to the process, mem file provides information about memory held by this process, status provides information about the memory and about the process, like the name, PID and so on.

Since the */proc/PID/maps* file provides which memory sectores are assigned to the process, can be made a dump data on this sectors of memory using the mem file. Considering that in the file */proc/PID/status* provides the name of the keyboard process and the PID assigned to it, can be made a scanning of each proc directory and read the status file in order to found which of them corresponds to the keyboard and make a memory dump. A tool is developed in order to make the memory dump to the keyboard process for search the keys pressed stored in the memory, the flowchart 1 describe how the tool works. However the results show that there are not a plain text in the memory about the keys we press in the keyboard. Continuing in the user space and considering that also when the driver is loaded into the kernel, it calls the function *input_register_device()* because it needs to indicate the creation of the file */dev/input/eventX* (where X is only an integer) that corresponds to the physical device. Is determined which *eventX* file corresponds to the touchscreen exploring the system, for analyze this file and determine if it can have information about the keys pressed.

The file corresponding to the touchscreen is the *event1*, the */proc/bus/input/devices* file provides this information. Trying to read the *event1* file we notice that it always is empty and only when an event occurs it has information but only for one instant. The Fig. 4 shows a hexadecimal representation of the data when the touchscreen is pressed, that is when the event occurs.

When an interrupt occurs the kernel needs to process it and with different functions in the include*/linux/input.h*, for instance *input_event(...)*, *input_report_abs(...)* and so on, the data is put in a standard format in the */dev/input/eventX* file in order to be accessed by the user space.

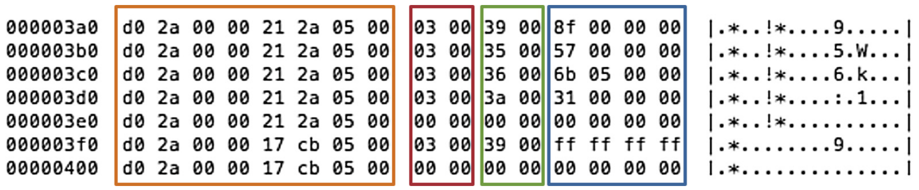
The *include/linux/input.h* provides information about the event standard format, it is a struct and has the next variables: time stamp, type, code and value [8], the Fig. 3 shows the code. Also *include/linux/input.h* file provides information about the meaning of each types and code values.



```
struct input_event{
                    struct timeval time;
                    __u16 type;
                    __u16 code;
                    __s32 value;
```

Fig. 3. Standard event format.

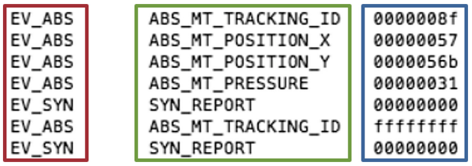
The hexadecimal data in Fig. 4 needs to be read from right to left for each hexadecimal value so as to understand them. The blue square are the values, for instance, the first value is *0000008f*, the green square are the codes, where according to */include/linux/input.h* the *0039* is the ABS_MT_TRACKING_ID which indicates the ID of the touch realized in that moment, the *0035* is the ABS_MT_POSITION_X which indicates the x coordinate of the touch, the *0036* is the ABS_MT_POSITION_Y which indicates the y coordinate of the touch, the *003a* is the ABS_MT_PRESSURE which indicates the pressure of the touch and *0000* is the SYN_REPORT which indicates the end of the report. The red square are the events type, where according to */include/linux/input.h* the *0003* means EV_ABS which indicates a touchscreen absolute event, and *0000* means EV_SYN which indicates a synchronize event. And finally the orange square indicates the timestamp.



```
000003a0  d0 2a 00 00 21 2a 05 00  03 00 39 00 8f 00 00 00  |.*..!*....9.....|
000003b0  d0 2a 00 00 21 2a 05 00  03 00 35 00 57 00 00 00  |.*..!*....5.W...|
000003c0  d0 2a 00 00 21 2a 05 00  03 00 36 00 6b 05 00 00  |.*..!*....6.k...|
000003d0  d0 2a 00 00 21 2a 05 00  03 00 3a 00 31 00 00 00  |.*..!*....:.1...|
000003e0  d0 2a 00 00 21 2a 05 00  00 00 00 00 00 00 00 00  |.*..!*..........|
000003f0  d0 2a 00 00 17 cb 05 00  03 00 39 00 ff ff ff ff  |.*........9.....|
00000400  d0 2a 00 00 17 cb 05 00  00 00 00 00 00 00 00 00  |.*..............|
```

Fig. 4. Reading the */dev/input/eventX* file with the hexdump command. Timestamps (orange square), events type (red square), codes (green square) and the values (blue square). (Color figure online)

With the command *getevent -l* the above interpretation in accord with the input.h documentation can be verify to be sure that the exploration was correct. This can be check with the Figs. 4 and 5.

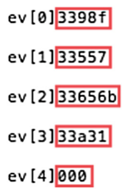


```
EV_ABS    ABS_MT_TRACKING_ID  0000008f
EV_ABS    ABS_MT_POSITION_X   00000057
EV_ABS    ABS_MT_POSITION_Y   0000056b
EV_ABS    ABS_MT_PRESSURE     00000031
EV_SYN    SYN_REPORT          00000000
EV_ABS    ABS_MT_TRACKING_ID  ffffffff
EV_SYN    SYN_REPORT          00000000
```

Fig. 5. Information showed with the getevent command. Events type (red square), codes (green square) and the values (blue square). (Color figure online)

A tool is made to experiment with this file with the purpose of get the information mentioned, because the file provides information about coordinates of the touches and can be used to determine if a key was pressed. The tool open the file */dev/input/event1*, and knowing the struct format, a same buffer struct needs to be indicate and the data can be acceded by specifying the elements in the struct in order to get the same data on the */dev/input/event1* file. The Fig. 6 shows how the software is able to take the event, key and value parameters. Due that in this file are given the coordinates, it could be used in order to make a keylogger, handling the data so as to get keys pressed.

Determine which touchscreen driver correspond to the device could be difficult because depends of different factors like the device itself and the kernel



```
ev[0]3398f

ev[1]33557

ev[2]33656b

ev[3]33a31

ev[4]000
```

Fig. 6. Extracting data from the */dev/input/eventX* file.

version, so it is difficult to be able to get information about the keys pressed as it difficult identify the current driver. Modify the kernel would allow to get information from the touchscreen's physical file and make it available to any app, like adding instructions to create a copy file without restriction permissions of the physical device file or even could be added a module in order to get data from the physical device file and put it in another file available to the apps, this options also depends in different factors like a device *rooted*, option module add enabled and user interaction. Getting information from the volatile memory also seems difficult for a malware, first because the malware needs get administrator privileges and second the data here is dynamic and the memory assigned to one process has a lot of data, make a software able to interpret this data and match some of these data with some specific characteristics and then get information will take a lot of resources and a malware doing this would make it simple to detect. However looks like that is possible extract information in the touchscreen's physical file only getting administrator privileges path due this has information about the touches coordinates.

Since there is a data flow to process the touches in the touchscreen is possible extract data about that touches, until the moment from one point, to maybe extract sensitive user information.

As we now have covered the kernel and a little the user space, the next is analyze the functions with a relation with the touchscreen and the keys pressed like *EventHub*, *InputReader*, *InputDispatcher* and so on.

5 Conclusion and Future Work

With the exploratory methodology is possible find points in the system where information about the keys pressed could be extracted, and making some experiments like examine and handling the data could be determined if the information has a relation with the keys pressed. This work shows that is possible get some information about the key pressed outside the Android application layer using the touchscreen physical file, and still without covering the full touchscreen's data flow, thus is possible that a malware is able to obtain information about the keys pressed without implement a third-part keyboard in Android.

For future work we are going to implement a tool that handle the data obtained from the *eventX* file in order to determine if is possible get user private information, we are going to analyze if is possible make some kernel modifications to provoke a information leak. Also we are going to continue analyzing the different functions related with the data flow and try to decide if there are points of information leak.

Acknowledgment. The authors are grateful to the Instituto Politcnico Nacional and the Consejo Nacional de Ciencia y Tecnologa by the support to this research.

References

1. Cho, J., Cho, G., Kim, H.: Keyboard or keylogger?: a security analysis of third-party keyboards on Android. In: 13th Annual Conference on Privacy, Security and Trust, pp. 173–176. IEEE (2015)
2. Mohsen, F., Bello-Ogunu, E., Shehab, M.: Investigating the keylogging threat in android—User perspective (Regular research paper). In: Second International Conference on Mobile and Secure Services (MobiSecServ), pp. 1–5. IEEE (2016)
3. Mohsen, F., Shehab, M.: Android keylogging threat. In: 9th International Conference on Collaborative Computing: Networking, Applications and Worksharing, pp. 545–552. IEEE (2013)
4. Kaspersky Lab.: What is a keylogger? http://www.kaspersky.com/au/internet-security-center/definitions/keylogger
5. Nasution, S.M., Purwanto, Y., Virgono, A., Ruriawan, M.F.: Modified kleptodata for spying soft-input keystroke and location based on Android mobile device. In: International Conference on Information Technology Systems and Innovation, pp. 1–5. IEEE (2015)
6. Hirabe, Y., Arakawa, Y., Yasumoto, K.: Logging all the touch operations on Android. In: Seventh International Conference on Mobile Computing and Ubiquitous Networking, pp. 93–94. IEEE (2014)
7. Damopoulos, D., Kambourakis, G., Gritzalis, S.: From keyloggers to touchloggers: take the rough with the smooth. Comput. Secur. **32**, 102–114 (2013)
8. Android open source project: Devices - Input. https://source.android.com/devices/input/index.html
9. Google Git: Git repositories on Android. https://android.googlesource.com/