MalDetec: A Non-root Approach for Dynamic Malware Detection in Android

Nachiket Trivedi and Manik Lal Das(✉)

DA-IICT, Gandhinagar, India
nachiket5197@gmail.com, maniklal_das@daiict.ac.in

Abstract. We present a malware detection technique for android using network traffic analysis. The proposed malware detection tool, termed as MalDetec, uses a non-root approach to notify the user in real-time about any malicious URL (Uniform Resource Locator) requests by any malware. MalDetec parses the packet dump file and merges it with the output of our network traffic analysis to generate App-URL pairing in real-time. This is later scanned by Virustotal databases and the user gets notified of suspicious URL requests. In addition, MalDetec maintains a local database containing results of previous scans for quick look-up during future scans. The experimental results show that MalDetec successfully detects the applications accessing malicious URLs, in real-time without having root privileges.

Keywords: Android · Malware detection · Virustotal
Network traffic analysis

1 Introduction

Android dominates the smartphone market by a large margin. According to Gartner [1], off all the phones sold in the last quarter of 2016, android accounted for 81.7%. With such a vast user base and fewer restrictions, security threat on phone device as well as applications is a growing concern. A recent survey by Nokia [2] states that during the second half of 2016, the increase in smartphone infections was 83% following on the heels of a 96% increase during the first half of the year, of which more than 80% was targeted for android.
Malware [3] is an executable code that has malicious intent to a computer, application or system. These malicious programs can perform a variety of functions, including stealing, encrypting or deleting sensitive data, hijacking core computing functions and monitoring users' activity without their permission. Many people save their private information like bank details, photos, passwords etc. on their smart phones, making all these information prone to such malware.
In this paper, we present a malware detection tool using network analysis in real-time. Many android malware contacts malicious servers and our aim is to detect such behaviors in real-time. We develop an application, MalDetec, which can successfully detect malicious URL requests using network traffic analysis [4].

R. K. Shyamasundar et al. (Eds.): ICISS 2017, LNCS 10717, pp. 231–240, 2017.
https://doi.org/10.1007/978-3-319-72598-7_14

MalDetec works in real time, getting URL scan results while the application runs in background, and does not require root privileges. Considering the fact that most of the android users do not root their phones, we made MalDetec completely independent of the device's root status. MalDetec works on the principle of capturing all the URLs accessed by different applications, scanning them for potential malicious content and notifying the user accordingly. We provide the complete working principles of MalDetec along with the experimental results. The remainder of the paper is organized as follows. Section 2 describes related work. Section 3 presents the proposed MalDetec tool. Section 4 provides analysis and experimental results. We conclude the work in Sect. 5.

2 Related Work

Malware [3,5] detection methods can be broadly classified into two parts: static and dynamic [6]. Static analysis or code analysis means the process of detecting malware when the malicious application is not being executed. This commonly includes segregating the different resources of the binary file without executing it and studying each component. Many smart malware obfuscate the malicious code and therefore, prove difficult to detect using normal static analysis. Dynamic analysis or behavioral analysis means studying the behavior of any application when it is actually running on a host system. Thus, the result of analysis can be obtained at the same instant the application is running. A popular method called sandboxing is used, which executes the malicious software in a virtual environment and thus, not having any adverse effects on the machine [7]. Many advanced malware can exhibit various evasive techniques specifically designed to defeat dynamic analysis including testing for virtual environments or active debuggers, delaying execution of malicious payloads, or requiring some form of interactive user input. Rastogi et al. [8] demonstrated this obfuscation method by using their tool DroidChamelon, which obfuscates malware and successfully evades known commercial anti-malware. There exists a few static and dynamic analysis detection methods in the literature. Song and Hengartner [9] proposed PrivacyGuard, a VPN based platform to dynamically detect information leakage on android devices. Chandramohan and Tan [10] demonstrated a wide variety of detection methods using both the approaches and explained them briefly. Isohara et al. [11] proposed a kernel based dynamic analysis method for malware Detection. Zaman et al. [13] proposed a static analysis method for malware detection in android based on network traffic analysis. Many a time, an application accesses a blacklisted URL while executing. For this, Zaman et al. suggested a method where the network packet data is collected from the android device for a particular time range and later analyzed on the computer. The URLs obtained are then scanned remotely by using the service provided by Virustotal and hence the scanned result is obtained. However, there are two shortcomings with Zaman et al.'s approach. First, the method captures the data from the device and analyses it later on an external computer, which can potentially infect the device before getting detected. It would be better if the tool analyzes

the packets dynamically and then, give the scanned results of the URLs as soon as the application accesses it. Second, the method used by them works only for a rooted device, which is risky, as one needs to give device's root privilege to the application.

3 MalDetec: The Proposed Malware Detection Tool

The proposed MalDetec tool analyses the network packets dynamically, scans the accessed URL for blacklist status without having the root privileges of the device, and reports the result to the user for suspicious actions of applications. Whenever an application sends an HTTP request, MalDetec forms a source port to timestamp to url mapping, which is done by continuous parsing of packet dump file (.pcap file) produced by a third party application named tPacketCapture. The other process gets the timestamp to port to app mapping and by merging the outcomes of the above two processes, we get the App-URL table. MalDetec also maintains a local database- hotdata, containing previous virustotal scan results. MalDetec first looks into the hotdata and if found, it reports the user if any blacklisted URL is accessed. If any app accesses a URL whose data is not present in the hotdata, a virustotal scan will take place and subsequently the hotdata will be updated. We also developed another application called Tester for testing purpose.

3.1 System Model

The participating entities in the whole process are android OS, the tPacketCapture [12] app, MalDetec, the uid-app database of MalDetec, the hotdata database of MalDetec, and various other applications which are opened and used by the user. Basically, whenever any application is opened and prompted to access any URL, it interacts with android OS to send HTTP request. Meanwhile, tPacketCapture interacts with android and by using its VPNService [14] it collects the packet dump. MalDetec runs synchronously and interacts with tPacketCapture to use the pcap file generated by it. Later on in the process, MalDetec interacts with its databases and also with android for its own network analysis and URL scanning.

3.2 Prerequisites and Assumptions

There are certain prerequisites for proper functioning of MalDetec. First and foremost, the tPacketCapture application should be installed and running before MalDetec is initiated. This is because MalDetec depends on tPacketCapture for packet dump generation. Another thing that is important but not necessary is to make a database of uid-app mapping, which is also considered in MalDetec. Excluding this step will not affect the working of MalDetec, but the application name will not be displayed when notifying the user. We also made an assumption that each application communicates with external servers via HTTP. Unless any

application communicates via a protocol(s) other than HTTP, MalDetec can successfully detect it. Furthermore, MalDetec depends on the services provided by Virustotal for examining the malevolent nature of any URL.

3.3 The Working Principles of MalDetec

We break the whole working of MalDetec in 5 parts as follows:

1. Design of MalDetec.
2. Capturing of packet dump and parsing .pcap file to get the time-port-URL mapping in real time.
3. Capturing the network statistics data and parsing it to get the time-port-App mapping in real time.
4. Merging the outcomes of steps (2) and (3) to get the App-URL table, followed by checking the URL in hotdata for scanned results.
5. Scan the URLs that are present in the for-later-scan database by using the virustotal API, notify the user as well as update the hotdata.

Design of MalDetec. We develop MalDetec using the Kivy framework [16] which is an open source project allowing developers to make android apps in python. It is based on the project python-for-android. We used python to make MalDetec instead of java because of various libraries provided by python for network analysis and parsing packet dump files, in addition to efficient and feasible file management. Another important reason of using Kivy framework was that the code just needs a slight modification to be compatible with iOS, which can be incorporated in the future. Before starting MalDetec, tPacketCapture app needs to be started for packet capturing. Once the main process is started, the working of the app is described in detail in Fig. 1.

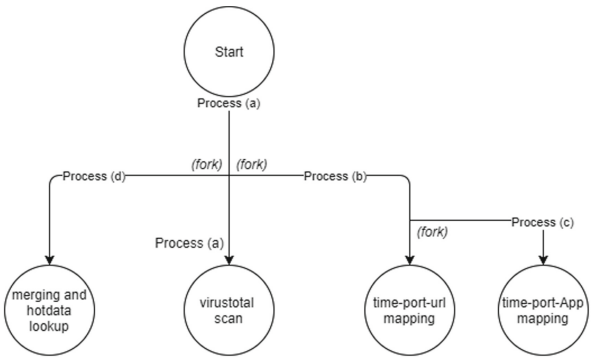


Fig. 1. Processes running concurrently in MalDetec

As shown in the Fig. 1, pressing the start button initiates the process-(a). This process is later forked into a child process-(b). This child process-(b) parses the

packet dump file generated by tPacketCapture and forks again to form another child process-(c) which generates netdump files to get time-port-app mapping. The initial parent process-(a) again forks to make another child process-(d) which is responsible for dynamic merging of the outcomes of process-(b) and process-(c) to make the App-URL table. Finally, the initial main process-(a) goes on to do a virustotal scan of the URLs that are generated from process-(d). All these processes are described in following sub sections.

Capturing of packet dump and parsing it to get the time-port-URL mapping. This phase involves continuous parsing of the packet-dump (.pcap) file which is produced by the tPacketCapture app. tPacketCapture is a third party application that captures all the network packets by using VPNService, a service provided by android, where it makes a vpn server for itself to capture network traffic. It makes a single .pcap file which contains all the network packet details and the process-(b) of Fig. 1 parses it. Here, we open the .pcap file to read and parse it in a continuous loop. This is done because tPacketCapture runs continuously in the background, and hence the packetdump file (.pcap file) gets updated constantly. As the .pcap file is opened and closed continuously, we made sure that the already parsed data of any previous iteration is not re-written in our final parsed file, thus saving the CPU time. For parsing the .pcap file, we used the dpkt framework [15] of python. The logic used in capturing of packet dump and parsing it to get the time-port-URL mapping is represented in Algorithm 1.

Algorithm 1. Parsing Packet Dump

```
1: while True do
2:     f = open(File,to read)
3:     pcap = dpkt.pcap.Reader(f)
4:     for ts,buf in pcap do
5:         eth = dpkt.ethernet.Ethernet(buf)
6:         ip=eth.data
7:         tcp=ip.data
8:         if tcp.dport=80 and len(tcp.data) > 0 then
9:             http = dpkt.http.Request(tcp.data)
```

First, we obtain the pcap object from the pcap file. By iterating through the pcap object, we derived the timestamp and the whole packet buffer for each packet. From the packet buffer, we got the Ethernet, IP and subsequently the TCP objects. The HTTP request data was obtained from the TCP object by dpkt only when TCP destination port was 80. Eventually, by parsing the HTTP object, we got the timestamp, URL, and port which was stored dynamically in a file. This process repeats itself till the user deliberately closes the app. The final file, in which the time-port-URL mapping is stored, should not be made from scratch in each iteration, and thus only the new packet data is parsed and appended to it in every iteration, thereby saving CPU time. Therefore, we get a dynamically generated file having the time-port-URL mapping from this step.

Capturing the network statistics data and parsing it to get the time-port-App mapping. With this phase, MalDetec maps the applications with the ports they were using at any particular time. To fulfill this task Zaman et al. [13] used the netstat command combined with busybox tool which enforced the user to have a rooted device. To overcome this problem, we used the /proc/net/ interface that does the job but does not require a rooted device. We used the proc interfaces [17] /proc/net/tcp and /proc/net/tcp6, which provide details of currently active TCP connections and are implemented by tcp4_seq_show() in net/ipv4/tcp_ipv4.c and tcp6_seq_show() in net/ipv6/tcp_ipv6.c, respectively. It first lists all listening TCP sockets, and next lists all established TCP connections. For using TCP protocol on IPv6 and IPv4, we used the tcp6 and tcp with the /proc/net interfaces. Therefore, by running the following command, a file was made which contains the command's output for a particular timestamp.

```
cat /proc/net/tcp6 >> /storage/emulated/0/MalDetec/files/dump/netdump\$i.txt
```

Now we execute this same command continuously after each second and hence derive a set of files, each containing the network statistics for a particular timestamp. When the above process keeps running, another process for merging these files runs concurrently. The process-(c) mentioned before in Fig. 1 is expanded as shown in Fig. 2.

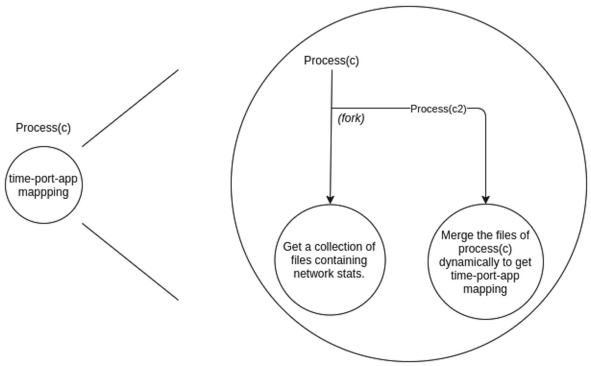


Fig. 2. A flowchart indicating the process-(c)

The other child process-(c2) that runs concurrently, merges all the files produced by the parent process-(c) into a single final file dynamically such that the contents of this file are the time-port-App mappings. Every second a new network statistics file is created as shown here. Thus, as soon as any new file is made, (c2) immediately parses it for the UID (a unique id is provided to each android application at the time of installation) of the app that was accessing a particular port at a particular time. The UID remains unchanged until the app is manually uninstalled (unlike the ProcessID). By getting the app package name from the UID, the time-port-App mapping is obtained and written in the

final file of this step. A major hindrance in this step was getting the app package name from the UID. A conventional way to do this task is to use the dumpsys tool [18] by the adb command.

```
adb shell "dumpsys package | grep -A1 ‘userId=%s’ "
```

This option works for a rooted device, as it needs access to application details for all apps on the device and it is present in /data/system/packages.list file, which is inaccessible without root permissions. To overcome this, we connected the phone with a computer and created an adb server. We ran this adb command for all the applications installed on the device and saved in the MalDetec app folder in the device, thus getting the UID-app package name mapping. This was done before MalDetec was started and is required to be done only one time per device. Therefore, while MalDetec is executing, during process-(c2), we extracted the app package name from its corresponding UID by grepping our local database. This way at the end, in the final file of this step, we obtain the time-port-app_package_name mapping dynamically.

App-URL table in real-time and scrutinizing the Hotdata. As the initial parent process-(a) forks into child process-(b) for network analysis, after some time period, another child process is created from the process-(a), depicted as process-(d) in the Fig. 1. The main function of this process is to dynamically merge the files created by process-(b) and process-(c) to get the final App-URL table and notify the user about the scan results of the URL if its data is present in the hotdata. Hotdata is a file that contains the URL scan results and details of previously scanned URLs. As virustotal only allows 4 requests per minute by using their public API, it becomes a very slow and inefficient method to scan every time. To overcome this we used hotdata which is a local database on the device that contains the details of previous scans. Therefore, in this step, after merging, every new URL in the merged file is concurrently searched for in the hotdata first. If found, the user is notified immediately of the details. If not found, the URL and the app name are written in another file for later virustotal scan explained below. Thus the use of hotdata increased performance efficiency of MalDetec.

URLs scanning process. The aforementioned process makes a file for all the App-URL mappings whose details are not present in the hotdata. This file is for later virustotal scan. Every *t minutes*, another process is invoked which scans these URLs by using the API provided by virustotal. Virustotal [19] is a service provided by Google, which uses about 64 URL scanners to scan any file/URLs. It provides developers an API key to use their public API to scan URLs or files, with a limitation of 4 requests per minute. The chosen format for the API is HTTP POST requests with JSON object responses. Therefore, as process-(d) dynamically updates the file *for-later-scan*, every *t minutes*, the new entries to that file is scanned by using virustotal API for potentially malicious content. It takes about 15 s to get the result of a particular request which is later notified to the user. The hotdata is also updated concurrently. The pseudo-code for this process is described in Algorithm 2.

The Request method sends a request to virustotal for the scan of the URL which is here encoded in the data parameter. The response from the virustotal is read by and the JSON file is extracted from it. From the JSON file we obtain a dictionary response_dict containing all the scan result details of the specified URL. All the information like positives, report link, scan date, total scanners can be obtained from this dictionary. The relevant information like the positives is right away notified to the user by a notification. These details are also updated in the hotdata. In the end, we obtain dynamic notifications for any malicious URLs accessed by any of the installed applications.

Algorithm 2. URLs scanning using Virustotal

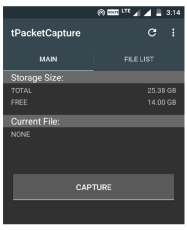
```
1: while True do
2:     f=open( for later scan file , to read )
3:     hdata=open( hotdata file , to append )
4:     for lines in f.readlines() do
5:         Api_Key = the api key
6:         words=lines.split()
7:         if len(words) = 2 then
8:             app=words[0]
9:             get_link=words[1] #url to scan
10:            url = 'https://www.virustotal.com/vtapi/v2/url/report'
11:            parameters = 'resource': get_link , 'apikey': Api_Key , 'scan':1
12:            data = urllib.urlencode(parameters)
13:            req = urllib2.Request(url,data)
14:            response = urllib2.URLopen(req)
15:            json = response.read()
16:            response_dict = simplejson.loads(json)
```

4 Experimental Results

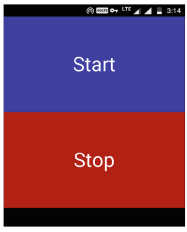
We first opened the tPacketCapture application and started to capture the packets. Then we started MalDetec and clicked start to initiate the process. As tPacketCapture and MalDetec were running in the background, we opened a series of applications having the potential to send a network request, one after another. In the background, tPacketCapture was collecting packetdump data into a .pcap file. MalDetec opens this .pcap file, parses it, and does its own network analysis to get the App-URL table and looks first in the hotdata. As soon the app amazon was opened, it downloaded a bunch of .png files whose report was present in our hotdata. Thus, a notification was sent to the user showing the details of the scan report as shown in the Fig. 3c.

For malware testing, we made another application, named Tester. Tester continuously accesses a bunch of URLs via HTTP some of which are known to have malicious data. Therefore, as soon as the Tester app started functioning, it began to send HTTP requests for those URLs in the background. MalDetec, being executed concurrently, catches the URLs and checks the hotdata first. Some of those

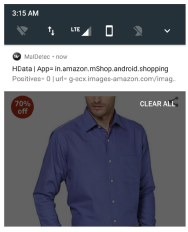
were not present in the hotdata and hence were placed in the for-later-scan file. As soon as the scan function was invoked after *t minutes*, a complete virustotal scan of the URLs present in the for-later-scan file took place. One such URL accessed by Tester was present in our hotdata. It was caught and scanned as positive by virustotal. Immediately the user was notified as shown below. Some malicious URLs were accessed after a while by the Tester app which were not present in our hotdata. In the notification, as depicted in Fig. 4, the user is notified that whether the URL accessed was present in hotdata (depicted as HData) or not (depicted as Vtotal). The user is even notified the number of positives a particular URL has according to virustotal and also the application name which accessed the URL. If at least one positive was detected according to virustotal, a notification with warning is sent to the user.



(a) tPacketCapture app starting page



(b) MalDetec: initiated by start button



(c) URLs from hotdata notified as HData

Fig. 3. Experimental screenshots

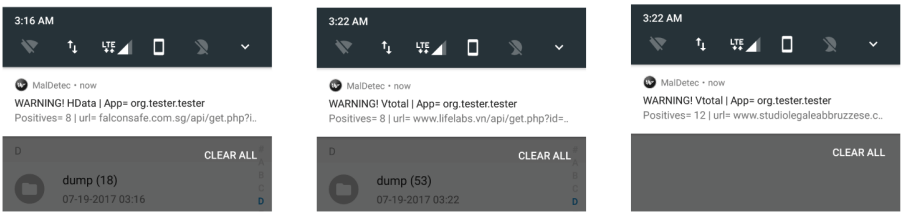


Fig. 4. Malicious URLs notified with warning along with the number of positives

5 Conclusions

We discussed a malware detection technique to dynamically detect any malicious URL access for any android device regardless of its root status. We proposed a tool- MalDetec which used network analysis to get the App-URL mapping in real time, scanned those URLs on the local database hotdata and notified the user accordingly. Whenever any URL which was already present in our hotdata was accessed, we were instantly notified. But, when the URL was not present,

it took a while for us to get the notification. This happened because MalDetec undertook a virustotal scan for those URLs and as virustotal only allowed 4 requests/minute, we got delayed results. Therefore, with a locally managed database, the user is notified in real-time of almost all the URL requests. We also demonstrated the experimental results of MalDetec. The limitation of MalDetec is that it cannot detect malware if the app accesses a URL via any protocol(s) other than HTTP, which is a potential future scope of MalDetec to extend its features further.

References

1. Gartner report 2017. http://www.gartner.com/newsroom/id/3609817
2. Nokia Threat Intelligence Report. Mobile infection rates rose steadily in 2016
3. Honig, A., Sikorski, M.: Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software. No Starch Press, San Francisco (2012)
4. Arora, A., Garg, S., Peddoju, S.K.: Malware detection using network traffic analysis in Android based mobile devices. In: Proceedings of International Conference on Next Generation Mobile Apps, Services and Technologies, pp. 66–71(2014)
5. Qian, Q., Cai, J., Xie, M., Zhang, R.: Malicious behavior analysis for Android applications. Int. J. Netw Secur. **18**(1), 182–192 (2016)
6. Distler, D.: Malware Analysis: An introduction. SANS Institute InfoSec, Reading (2007)
7. CWSandbox Automates Malware Analysis. http://www.securitypronews.com/cwsandbox-automates-malware-analysis-2006-10. Accessed July 2016
8. Rastogi, V., Chen, Y., Jiang, X.: DroidChameleon: evaluating android anti-malware against transformation attacks. In: Proceedings of ACM SIGSAC Symposium on Information, Computer and Communications Security, pp. 329–334 (2013)
9. Song, Y., Hengartner, U.: PrivacyGuard: a VPN-based platform to detect information leakage on Android devices. In: Proceedings of ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices, pp. 15–26 (2015)
10. Chandramohan, M., Tan, H.: Detection of mobile malware in the wild. IEEE J. Comput. **45**(9), 65–71 (2012)
11. Isohara, T., Takemori, K., Kubota, A.: Kernel-based behavior analysis for Android malware detection. In: Proceedings of International Conference on Computational Intelligence and Security, pp. 1011–1015 (2011)
12. tPacketCapture. http://www.taosoftware.co.jp/en/android/packetcapture/
13. Zaman, M., Siddiqui, T., Amin, M.R., Hossain, S.M.: Malware detection in Android by network traffic analysis. In: Proceedings of International Conference on Networking Systems and Security (2015)
14. VpnService. https://developer.android.com/reference/android/net/VpnService.html
15. J Oberheide. Parsing a PCAP file. https://jon.oberheide.org/blog/2008/10/15/dpkt-tutorial-2-parsing-a-pcap-file/. Accessed Jan 2017
16. Kivy - Open source Python library. https://kivy.org/. Accessed Jan 2017
17. /proc/net/tcp documentation. https://goo.gl/2TVZNp. Accessed Jan 2017
18. DUMPSYS: Tool to get system services details by ADB. https://developer.android.com/studio/command-line/dumpsys.html
19. VirusTotal. https://www.virustotal.com