Detection of Android API Call Using Logging Mechanism within Android Framework

Yuuki Nishimoto[1,⋆], Naoya Kajiwara[2,3], Shinichi Matsumoto[2,3], Yoshiaki Hori[3,4], and Kouichi Sakurai[2,3]

[1] Department of EECS, Kyushu University, Fukuoka, Japan
[2] Department of Informatics, Kyushu University, Fukuoka, Japan
kajiwara@itslab.inf.kyushu-u.ac.jp,
sakurai@csce.kyushu-u.ac.jp
[3] Institute of Systems, Information Technologies and Nanotechnologies
smatsumoto@isit.or.jp
[4] Organization for General Education, Saga University, Saga, Japan
horiyo@cc.saga-u.ac.jp

Abstract. Android based smartphones have become popular. Accordingly, many malwares are developed. The malwares target information leaked from Android. However, it is difficult for users to judge the availability of application by understanding the potential threats in the application. In this paper, we focus on acquisition of information by using a remote procedure call when we invoke the API to acquire phone ID. We design a methodology to record invocation that are concerned the API by inserting Log.v methods. We examined our method, and confirm empirically the record of the call behavior of the API to acquire phone ID.

Keywords: Android, Malware, Privacy Protection, Dynamic Analysis.

1 Introduction

In recent years, Android phone is becoming popular. Simultaneously, many malicious applications called malware are developed for Android platform. Many malwares that target Android cause information leakages, and leakage of personal information is a big problem. However, it is difficult for a user to grasp threats of an application and judge the risk of it. Therefore, we focus on a measure method to prevent malwares from being distributed in the marketplace. In this method, an application developer and a marketplace operator can previously check the application on behalf of the user.

There are dynamic analysis and static analysis in approach of malware detection. However, in static analysis, there is a possibility that overlooking increases when the variants of the malware outbreak. In dynamic analysis, there are some problems too. With dynamic analysis, overhead of operation increases. Moreover,

⋆ The first author currently works with Kyushu District Police Bureau. He contributed to this research when he was a undergraduate student of Kyushu University.

T. Zia et al. (Eds.): SecureComm 2013, LNICST 127, pp. 393–404, 2013.
© Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2013

there is a possibility that a malicious developer can make his application to circumvent the detection. In order to solve these problems, we focus on detection method using log output which is dynamic analysis. With marketplace operator using this method, detection which cannot be circumvented by malicious developer can be realized.

Linux debug utility named `strace` monitors system calls used by an application in Android. There is a method performing malware detection by analyzing system calls that are obtained using the `strace`. Behavior using services of the kernel can be detected by this method. However, there is a problem that system call is not issued in the behavior which doesn't use services of kernel, and it is impossible to detect such a behavior.

In this paper, we focus on the fact that when API that retrieve the phoneID is invoked, it is processed with remote procedure call. We propose a method recording the invocations in accordance with the API by inserting `Log.v` method. That is an output log API in remote procedure call by the Android Binder. The execution logging by this technique cannot be avoided even if modification of API is performed on the call side. Therefore, it is impossible to circumvent the detection even if a developer has malicious intention. Furthermore, we implemented proposal method tentatively and ran the application which acquires phoneID on the Android emulator. As a consequence, we confirm empirically record of invocation behavior of the phoneID acquisition API.

2 Android

Android is a platform that was developed targeting mobile information devices such as smart phones and tablet PCs. Android application is running on the Dalvik virtual machine(VM). When an Android application is launched, one Dalvik VM is dedicated to execute that application. When a user installs an application, by approving permissions, it becomes possible to take the cooperation with other applications or access files made by other applications deviating from the sandbox mechanism[1].

2.1 Android Application

Android consists of a Linux based OS kernel, a middleware and fundamental applications. The applications used by the user are on the top layer in AndroidOS. Developers can publish applications in the Android market.

In the application framework of Android, API is provided and Android application developer can use it freely. Application developers can use the result the API outputs by this mechanism, without knowing about the complicated procedures under the framework layer.

In AndroidOS, the independence of applications is maintained by the following mechanisms so that applications don't cause interference.

- Application execution and process
  Android application is executed in an individual Linux process allocated to this sole application. Hence, a Linux process is started when an application is executed. However, this process is terminated when system resources are required from other applications, after this application is finished.
- Dalvik virtual machine allocation to every process
  In Android, a Dalvik virtual machine is allocated to every process exclusively. In this way, one application is executed independently from other applications.
- Unique Linux user ID allocated to every application
  During the installation, to every application is allocated unique ID. The assigned ID serves as an owner of the application, and manages the process. Files that the application creates are set up so that these files cannot be fundamentally read from applications which have other ID. For this reason, a file created by a certain application cannot be freely read from an application with another ID.

2.2 Binder

Binder is a driver which offers the functionality to communicate between processes. Even if some processes are in the same application, they run on separate area. Moreover, there is a possibility that activities and services respectively run on different processes in the application. Binder driver is used when exchanging information between these different processes. In this case, communications are controlled by the framework layer located above the kernel. Although a user doesn't use Binder directly, it plays an important role in interprocess communication.

2.3 AIDL

In Android, one process cannot usually access memory of other processes. Therefore, if a process wants to obtain data from other processes, interprocess communication is necessary. AIDL(Android Interface Definition Language) is an interface definition language used to generate some codes[2]. These codes undertake the communication between two processes possible using interprocess communication realized with Binder.

2.4 Android API

API, which means "Application Programming Interface", is an interface to access function from the library intended for the OS and for the programming language of the applications. Functions used in many applications are offered within the application framework of Android through API. Because it is unnecessary to develop functions offered by API, development of application becomes easy with API. Some of Android APIs also offer functions which serve as a base of the OS.

3 Existing Android Malware and Detection Method

There are static analysis and dynamic analysis among analysis methods of applications. Static analysis is a method that an application is decompiled and source code is examined, and dynamic analysis is a method that analyzes the behavior of an application by running it. There are both advantages and disadvantages for static analysis and dynamic analysis, and it is difficult to say which method is better than the other. In this section, we outline static analysis and dynamic analysis, as well as detection technique of malwares that both techniques use.

3.1 Android Malware

Malware is an application which performs a malicious action, such as causing a leakage of privacy information or making data destroyed. Malware is developed according to an environment with many targets. Therefore, malwares for Windows with many users have accounted for a large percentage of entire malwares until now. However, developers of malware also came to target AndroidOS. According to G Data Malware Report -Half yearly report January - June 2011 -[3], during the first half of 2011 from the second half of 2010, malwares that target smartphones with a focus on Android had increased from 55 to 803. Although this number is lower than the number of malwares which target Windows, considering the kind of information stored in Android devices is important personal information such as phone number or subscriber ID, it is thought that the risk from a security point of view becomes high compared with other OS. From these facts, despite enhancements of security including anti-malware in AndroidOS is in urgent, the present security is insufficient. Most threats caused by Android malwares are infection by installing the malwares that are obtained from a third party market that is not legitimate Android market of Google.

3.2 Static Analysis

Static analysis is a program analysis method which analyzes a program by decompiling the application without performing an executable file. Static analysis is mainly used when analyzing a source code.

Because analysis is performed without actually executing the application, the potential threat is detected before the damage of malware occurs. On the other hand, when the source code of the application is obfuscated or when the code for attack is placed outside the application code using a cooperation function with an external server, the possibility of being undetectable becomes high.

As static analysis method, there are many certification techniques and such techniques are served as service. Bouncer[4] is a service offered by Google. It prevents malware from spreading the market. However, malwares which have passed bouncer's certification had been reported[6].

3.3 Dynamic Analysis

Dynamic analysis is a program analysis method which checks what kind of action the application is carrying out by actually executing the application to be inspected. Because application is actually run unlike static analysis and it is inspected based on the action, malwares can be detected even when the source code is obfuscated, or when the code for an attack is placed outside the application code.

TaintDroid[5] and AppFence[7] are dynamic analysis methods using information flow tracking. TaintDroid monitors interprocess communications, and if information is sent out TaintDroid alerts that event. Appfense implements two information protections, replacing private data with shadow data and filtering to prevent information leakage by intercepting the network system call. Both researches modify Android kernel to conduct dynamic analysis for applications.

A logging system is used as a way of dynamic analysis. Isohara et al. proposed a logging system in Android[8]. System calls are collected as log data in the kernel level. These log data are analyzed with signature of threats to inspect the application's behavior. However, a problem is that action without system call is difficult to detect.

4 Design of Record Method of Process Operation Using Logcat

4.1 Record Method of Process Operation

`strace` is a debugging utility that supervises the system calls issued by a program. In the process action recording method using this `strace`, the system call about the API cannot be recorded if the API is belonged to `TelephonyManager` class, This is because the information is called without using a service of the kernel, when using API of a `TelephonyManager` class.

We insert a code that invokes `Log.v` method into the application framework of AndroidOS, and modify it so that event logs may be output. And when an application acquires the phoneID through API, the event log is recorded, and we use a method of performing detection of information retrieval based on that log. This method is also used in other OS. For example, in UNIX OS, a log is recorded using `syslog`, and in Windows, a log is recorded using the function named event log.

APIs which record logs are prepared within the application framework of AndroidOS. These logs can be viewed using the function called `logcat`. In this experiment, logs are collected and analyzed, which are output from a `Log.v()` method of the `Log` class. This method is implemented in the layer which uses Java language in application framework.

We examined interprocess communications which occur when using APIs of `TelephonyManager` class. As a result, it turned out that processes and methods communicate in the procedure as shown in a Figure 1.

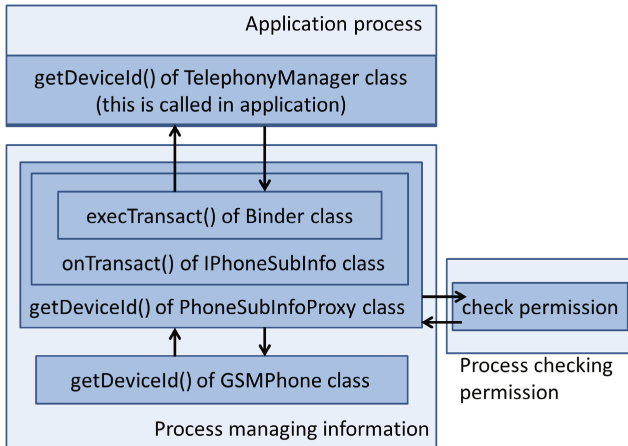

Fig. 1. Example of interprocess communication about getDeviceId()

The approach of checking which method is invoked by making log output is a general technique performed in other OS. When performing one application in Android, it is always run on independent Dalvik VM. Therefore, an application cannot communicate with other processes directly, and the application must use a driver called Binder. Then, we set a code that outputs a log in programs which perform this interprocess communication, and when an application invokes API, we detected and specified it based on the log information. This is the new point in this proposal method. With this method, retrieved information can be checked by seeing .aidl file without searching for the part which reads each information directly. In this experiment, after an application is executed, invoked API can be specified using the information acquired from the event log.

In this paper, the experiment was carried out for API contained in the `IPhoneSubInfo.aidl` file treating important information such as telephone number or subscriber ID. Concretely, at first the Log class of an `android.util` package is imported to `IPhoneSubInfo.java` file. Then, the code which outputs a log to an `onTransact` method is inserted. OS is recompiled after that. Application which invokes some APIs is installed to the emulator, and it is actually executed. From obtained event logs, we focus on the variable named `code` used in `onTransact`.

Table 1 shows the conversion table of the information about API contained in an `IPhoneSubInfo.aidl` file and each API. In this paper, experiments are not carried out for `getLine1AlphaTag()` and `getCompleteVoiceMailNumber()`. The reasons are the following two.

– In spite of being implemented in `TelephonyManager.java`, these two methods are undocumented as methods of the `TelephonyManager` class in the site of Android Developer.

Table 1. API defined in IPhoneSubInfo.aidl

API	Acquired Information
`getDeviceId()`	IMEI
`getDeviceSvn()` (Method within `getDeviceSoftwareVersion()`)	Software version of device
`getSubscriberId()`	Subscriber ID
`getIccSerialNumber()` (Method within `getSimSerialNumber()`)	Serial number of SIM card
`getLine1Number()`	Phone number
`getLine1AlphaTag()`	Alpha identifier
`getVoiceMailNumber()`	Voice mail number
`getCompleteVoiceMailNumber()`	Complete voice mail number
`getVoiceMailAlphaTag()`	Voice mail alpha identifier

– If we try to use these methods as methods of a `TelephonyManager` class in a application, the error message that it is undefined within `TelephonyManager` will come out.

From these reasons, experiments are carried out for seven APIs except the previously mentioned two.

4.2 Abstract of Experiment

The goal of these experiments is not the static analysis that decompiles application and analyzes a source code but the dynamic analysis that detects information leakages by actually running the application and taking event logs. In order to prevent from being detected by anti malware software, recent malwares obfuscate itself to make such an analysis difficult, or cause information leakages in cooperation with external server using `webkit`. The reason for using dynamic analysis in this paper is because it can deal with situations that static analysis cannot.

From the result of the record method using `strace`, it is predicted that personal information acquired by APIs of `TelephonyManager` is not retrieved by the kernel, but passed from other information managing processes. So, we focus on Binder driver which has an important role in interprocess communication. In this paper, we carried out the experiment which detects that event when APIs described in `IPhoneSubInfor.aidl` are invoked. We inserted a code that invokes `Log.v` methods which outputs a log message into `onTransact` method in `IPhoneSubInfo` class invoked only when these APIs are invoked. Then, we tried to specify the invoked API from the event logs. An argument called `code` exists in `onTransact` method of `IPhoneSubInfo` class. `OnTransact` method judges which API invoked information from this `code` value. Therefore, we think that we can specify which API is invoked from the event log of `onTransact` method and `code` variable.

```
@Override public boolean onTransact(int code, android.os.Parcel data, android.os.Parcel reply,
int flags) throws android.os.RemoteException
{
Log.v("EXTEST","IPhoneSubInfo.onTransact, code:" + code);
switch (code)
{
case INTERFACE_TRANSACTION:    code outputting log
{
reply.writeString(DESCRIPTOR);
return true;
}
case TRANSACTION_getDeviceId:
{
data.enforceInterface(DESCRIPTOR);
java.lang.String _result = this.getDeviceId();
reply.writeNoException();
reply.writeString(_result);
return true;
}
......
............
```

Fig. 2. Log outputting code inserted into IPhoneSubInfo.java

4.3 Proposal Method

Figure 2 shows a inserted code outputting a event log into `IPhoneSubInfo.java`. The place where a logging code is inserted was decided in consideration of the following conditions.

- It is not a method performed in the same process as an application.
  The `getDeviceId()` method of `TelephonyManager` class is run in the same process as the application. Such a method can be incorporated as a library in application by developer when application is developed. In the case of inserting the code which outputs the event log into `getDeviceId()` method, if a malware developer defines a method working similarly as `getDeviceId()` in that application and executes the method, information is retrieved without outputting the log. Therefore, it is very important to insert the code which outputs the log into a method running on a process which is not same as the application's process.
- API which invoked the method can be specified
  Even if a log is output from the code inserted into the program, this method is not realized if which API was invoked by the application cannot be checked from the log.
- Proposal method can be used to detect as many APIs as possible
  If the code which outputs a log is inserted into each API method (e.g. `getDeviceId()`), only that information can be monitored.

As a result of considering these three conditions, we concluded that a suitable inserted place should be `onTransact` method of `IPhoneSubInfo` class. There are some reasons for this decision. `onTransact` method doesn't run on the same process as application. Then, because the order of methods defined in

IPhoneSubInfo.aidl file and value of code variable are corresponded, it is possible to judge which API was invoked. This is because code variable is used within a switch statement in IPhoneSubInfo class. Furthermore, nine APIs defined in IPhoneSubInfo.aidl can be inspected with this method.

There is also an advantage that APIs which don't issue system call can be detected. In existing research, the detection of malware is performed by logging system call when using dynamic analysis[8]. However, in such a method, it is difficult to detect APIs which don't publish system call when running. On the other hand, because we focus on interprocess communication which occurs when the API is invoked, and insert a code which outputs the log when method is invoked, it becomes possible to realize detection of information retrieval without system call.

4.4 Experimental Procedure

1. Building of the source code
   The make command is used to build the source code. IPhoneSubInfo.java file is automatically generated from IPhoneSubInfo.aidl file at this time.
2. Insertion of a code which outputs a log
   Figure 2 shows modified IPhoneSubInfo.java to output the event log. This program outputs a log message which can be seen with Logcat view. Log.v is a method which outputs a log of a detailed message. There are other methods about log. Log.e outputs a log about error, Log.w outputs a log of warning, Log.i outputs a log about information, and Log.d outputs a log of the debug message. Fundamentally, usage of these methods is the same. String indicating tag is set as first argument and String which should be output as a log message is set as second argument. The differences among these five methods are found in use, and they are properly used so that acknowledgement of logs becomes convenient. In this experiment, a log message includes two contents. First content is a character string called IPhoneSubInfo.onTransact. And, second content is a value of code variable, which indicates a kind of privacy information acquired by an application.
3. Rebuild
   After rewriting and saving IPhoneSubInfo.java file, build is performed again.
4. Running application on the emulator
   This experiment is entirely conducted on the emulator.
5. Reference of logs
   Collected Logs are referred using Dalvik Debug Monitor Service tool. We describe and considerer the results from these collected logs.

4.5 Result

A kind of information acquired by API could be detected from output logs. Table 2 shows the correspondence of APIs used in experiment to code variables. This corresponds with the order of method defined in IPhoneSubInfo.aidl file.

Table 2. The correspondence table of API used in the experiment and code variable

code	API
1	`getDeviceId()`
2	`getDeviceSvn()` (Method within `getDeviceSoftwareVersion()`)
3	`getSubscriberId()`
4	`getIccSerialNumber()` (Method within `getSimSerialNumber()`)
5	`getLine1Number()`
6	`getLine1AlphaTag()`
7	`getVoiceMailNumber()`
8	`getCompleteVoiceMailNumber()`
9	`getVoiceMailAlphaTag()`

This shows that we can know `code` variables corresponding to each method from AndroidOS source code.

Figure 3 shows that logs output when `getDeviceId()` method is executed. The emphasized line in Figure 3 is a log message which outputs the necessary information in our proposal method.

4.6 Consideration

This experiment showed that detection and specification of API invoked from application can be possible. In this experiment, we inserted the code which outputs a log into `onTransact` method of `IPhoneSubInfo` class because we focus

```
PID  Message
335 getDeviceId_begin
335 TelephonyManager.getDeviceId
147 Binder.execTransact_in
147 Binder.execTransact_try_in
147 IPhoneSubInfo.onTransact, code:1
147 PhoneSubInfoProxy.getDeviceId
147 PhoneSubInfo.getDeviceId-permission_before
147 ContextWrapper.enforceCallingOrSelfPermission
147 ContextImpl.enforceCallingOrSelfPermission
147 ContextImpl.checkCallingOrSelfPermission
74  Binder.execTransact_in
74  Binder.execTransact_try_in
74  ActivityManagerService.onTransact
74  ActivityManagerNative.onTransact
74  Binder.execTransact_try_out
74  Binder.execTransact_out
147 PhoneSubInfo.getDeviceId-permission_after
147 GSMPhone.getDeviceId:000000000000000
147 Binder.execTransact_try_out
147 Binder.execTransact_out
335 getDeviceId_done
```


Fig. 3. Log output when executing getDeviceId() method

on APIs defined in `IPhoneSubInfo.aidl`. It is thought that APIs which are not mentioned in this paper are also defined in aidl file if they execute interprocess communication. Therefore, action of API is detectable by discovering the aidl file and conducting the same experiment as this one.

5 Conclusion

In this paper, a detection method of phoneID acquisition using `logcat` is proposed. With this method, it is possible to detect obfuscated applications which cannot be detected with static analysis, or phoneID acquisition of an application which sets attack code in an external server. The phoneID acquisition of API which cannot be detected with dynamic analysis using `strace` could be detected. Because we focus on the behavior of applications in our method, it is unnecessary to acquire signatures of malwares in advance. Therefore, unknown malwares can be detected with proposal method. Moreover, the system which outputs the log in the proposal method is completely independent of the structure of application thanks to the mechanism which retrieves phoneID as shown in Figure 1. For this reason, a malicious developer is unable to avoid the analysis by this technique.

In a practical use, the proposal method should be used by marketplace operator. One of the reasons is that the proposal method has no real-time properties. The proposal detection method needs to be performed before a user runs an application on his device because the method grasps the behavior of an application from the log output. Another reason is that the proposal method needs to rebuild AndroidOS and to prepare linux system for the analysis. From these reasons, the proposal method should be used in the marketplace operator's side.

As for future work, the distinction between malwares and legitimate applications is considered. This method detects all applications that acquire phoneID through API on the characteristics. When actually used, it is necessary to extract only malware from these applications and specify it. In this paper, we carried out experiments only about API defined in `IPhoneSubInfo.aidl`. However, we didn't carry out experiments about other APIs. As a future subject, we must confirm if proposal method can be applied to detection of other APIs.

Acknowledgments. We would like to thank Ayumu Kubota and Takamasa Isohara, KDDI R&D Labs for giving beneficial advices. This work was supported by Grants-in-Aid for Scientific Research (B)(23300027), Japan Society for the Promotion of Science (JSPS).

References

1. Permissions — Android Developers, http://developer.android.com/guide/topics/security/permissions.html
2. Android Interface Definition Language (AIDL) — Android Developers, http://developer.android.com/guide/components/aidl.html

3. Ralf Benzmüller, Sabrina Berkenkopf: G Data Malware Report Half-yearly report January [June 2011, http://www.gdatasoftware.co.uk/uploads/media/G_Data_MalwareReport_H1_2011_EN.pdf
4. Android and Security, http://googlemobile.blogspot.jp/2012/02/android-and-security.html
5. William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. , "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," In Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI), October, 2010. Vancouver, BC.
6. Dissecting Android's Bouncer, https://blog.duosecurity.com/2012/06/dissecting-androids-bouncer/
7. Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, David Wetherall, "These Aren't the Droids You're Looking For: Retrofitting Android to Protect Data from Imperious Applications," In Proceedings of the 18th ACM Conference on Computer and Communications Security, October, 2011.
8. Takamasa Isohara, Keisuke Takemori, Ayumu Kubota, "Kernel-based Behavior Analysis for Android Malware Detection," In Proceedings of the 7th International Conference on Computational Intelligence and Security, December, 2011.