Data Leakage in Mobile Malware: the what, the why and the how.

Corrado Aaron Visaggio[a], Gerardo Canfora[a], Luigi Gentile[b], Francesco Mercaldo[c]

[a]*Department of Engineering, University of Sannio, Benevento, Italy*
[b]*Koine srl, Benevento, Italy*
[c]*Institute for Informatics and Telematics, National Research Council of Italy (CNR), Pisa, Italy*

Abstract

Mobile technologies are spreading at a very quick pace. Differently from desktop PCs, smartphones, tablets and wearable devices, manage a lot of sensitive information of the device's owner. For this reason they represent a very appealing opportunity for attackers to write malicious apps that are able to steal such information. In this paper we analyse a huge set of Android malwares in order to discover which kind of data is exfiltrated from mobile devices and which are the mechanisms that malware writers leverage. For this analysis three tools were employed which are considered the state of the art of the available technology: Flowdroid, Amandroid, and Epicc. Our results show that mobile malware usually exposes users to a massive data leakage.

Keywords: security, testing, privacy

1. Introduction

According to the 2016 Internet Security Threat Report released by Symantec, the volume of new malware for mobile devices is dramatically growing: the number of Android variants increased by 40 percent in 2015, compared with 29 percent growth in the previous year.

The combination of rich sensors and ubiquitous connectivity makes smartphones the perfect candidates for privacy attacks. It is a consolidated habit of apps writers to write code for tracking users and leaking their personally identifiable information [1, 4, 5, 7], while users are generally unaware and unable to contrast them[2, 6]. In fact, the only real defense for sensitive data is the user that should deny those permissions that request the usage of sensitive data. However Android architecture does not provide for a mechanism to signaling the user which app is using which data and if data are transmitted to a third party.

By exploiting this mechanism and the scarce attention of users to this problem, many apps share sensitive user data with third parties [8], without warning or acknowledging the user about that. Dynamic loading makes worse the situation, as a fragment of code that

Email addresses: visaggio@unisannio.it (Corrado Aaron Visaggio), canfora@unisannio.it (Gerardo Canfora), digitangi@gmail.com (Luigi Gentile), francesco.mercaldo@iit.cnr.it (Francesco Mercaldo)

steals sensitive information could be loaded and executed by an app after the scanning of an antimalware.

One of the main goals of the malware targeting mobile devices is to steal sensitive information, by sending it to a drop server or a remote controller. This kind of data is mined by cyber-criminals for many reasons: identity theft, scams, phishing, harassment.

Considered how easily and how frequently data is stolen by malware in Android devices, it is needed to strenghten the mechanisms for protecting sensitive data. In order to design such mechanisms, it is necessary to understand in detail which are the methods used by malware for gathering sensitive data.

In this chapter we analyze which are the processes and the techniques used by malware for capturing sensitive data, which kinds of sensitive data are collected, which are the most common code patterns used, and where sensitive data are sent after having been gathered.

In order to realize this study, we analyzed the data leakage implemented by 4,593 malware, belonging to 11 malware families, obtained by official datasets or repositories releasing malware samples. Furthermore, the study demonstrates how widespread among malwares are those functions related to data leakage.

Usually two kinds of data leakage can be accomplished: one between applications and another one consisting in the shipping of (sensitive) data exfiltrated from a target device to a third party server or destination that is external to the target device. Our study examines only the latter case.

Evidence about data leakage is extracted by using three tools that are considered the state of the art for this kind of analysis: FlowDroid [1], Amandroid [2] and Epicc [3]. The three tools are able to collect complementary pieces of information: Flowdroid identifies the overall set of connections between the sources and the sinks involved in the data exfiltration; Amanandroid extends some features of Flowdroid, capturing the pattern of actions that lead to the data theft; and, finally, Epicc retrieves explicit ad implicit intents that could be leveraged for extracting sensitive data.

The reminder of the paper is organized as follows: Section 2 thoroughly analyses related work, Section 3 introduces the tool chain used for the analysis, and Section 4 illustrates the results of the experiment. Finally, Section 5 draws the conclusions.

2. Related Work

The problem of detecting data leakage has been faced in literature from three main perspectives. The first one deals with the code patterns that exfiltrate sensitive information and send it to a third party. The second one is the data leakage between apps, that occurs when an app allows third party apps to intercept its methods that handle sensitive data. The third perspective concerns the detection of malicious software that is able to steal sensitive data.

ScanDal [4] is a static analyzer performing both flow-sensitive and flow-insensitive analysis of an application for detecting privacy leaks in Android applications. To detect privacy leak, ScanDal collects information on where the value was created. When a value is created at an information source, ScanDal denotes the program counter of the source and sends it

to the analysis. When a value is created from existing values, ScanDal denotes the union of all the sets of the program counters from existing values. By this, ScanDal can collect every values which could be created from information sources. If such values flow out through an information sink, ScanDal detects it and considers it as a privacy leak. Authors analyzed 90 popular applications from Android Market and detected privacy leaks in 11 applications. They also experimented 8 known malicious applications from third-party markets and detected privacy leaks in all the 8 applications.

TaintDroid [5] is an extension of the Android operating system that tracks the flow of privacy sensitive data through third-party applications. TaintDroid assumes that downloaded, third-party applications are not trusted, and monitors–in realtime–how these applications access and manipulate users' personal data. TaintDroid labels data from privacy-sensitive sources and transitively applies labels as sensitive data propagates through program variables, files, and interprocess messages. The authors using TaintDroid monitored the behavior of 30 popular third-party Android applications, finding 68 instances of potential misuse of users' private information across 20 applications.

IccTA [6] uses a static taint analysis technique to find privacy leaks, e.g., paths from sensitive data, called sources, to statements sending the data outside the application or device, called sinks. A path may be within a single component or cross multiple components. To verify their approach, authors developed 22 apps containing ICC-based privacy leaks. Another tool proposed by the scientific community is CHEX [7], that uses static analysis to detect component hijacking vulnerabilities in Android applications by tracking taints between sensitive sources and externally accessible interfaces. However, it is limited to at most 1-object-sensitivity which leads to imprecision in practice.

PCLeaks [8] performs data-flow analysis to detect potential component leaks, which detects both component hijacking vulnerabilities and component launch (or injection) vulnerabilities. ContentScope [9] is another tool that tackles potential component leaks. Authors exclusively focus on the open content provider interface of Android apps and study potential risks that may lead to passive privacy leakage and unintended manipulation of security-sensitive data.

Papamartzivanos and colleagues [10] provide a solution for real-time tracking the privacy-flow of a user. Futhermore, they develop a collaborative infrastructure for exchanging information related to apps' privacy exposure level, and a behavior-driven detection mechanism in an effort to take advantage of the crowdsourcing data to its maximum efficacy.

Amandroid [2] performs an Inter-Component Communication (ICC) analysis to detect data leakage between apps, and has been developed concurrently with IccTA. Amandroid builds an Inter-component Data Flow Graph (ICDFG) and a Data Dependence Graph (DDG) to perform ICC analysis. Amandroid provides a general framework to enable analysts to build a customized analysis on likely data leakage between Andorid apps.

FlowDroid [1] detects those apps that permit data leakage. It helps to reduce the leaks or false positives. Novel on-demand algorithms allow FlowDroid to maintain efficiency despite its strong context and object sensitivity.

Epicc [3] identifies a specification for every ICC source and sink. This includes the location of the ICC entry point or exit point, the ICC Intent action, data type and category,

as well as the ICC Intent key/value types and the target component name. Note that where ICC values are not fixed, Epicc infers all the possible ICC values, thereby building a complete specification of the possible ways ICC can be used. The specifications are recorded in a database as flows detected by matching compatible specifications.

ComDroid [11] is able to detect application communication vulnerabilities; differently from other approaches, it analyses Dalvik bytecode. The tool is able to examine inter-application communication and present several classes of potential attacks on applications. Outgoing communication can put an application at risk of Broadcast theft (including eavesdropping and denial of service), data theft, result modification, and Activity and Service hijacking. Incoming communication can put an application at risk of malicious Activity and Service launches and Broadcast injection. Authors analyzed 20 applications and founded 34 exploitable vulnerabilities; 12 out of the 20 applications have at least one vulnerability.

DidFail [12] leverages FlowDroid [1] and Epicc [3] to detect ICC leaks. Currently, it focuses on ICC leaks between Activities through implicit Intents. Thus, it will miss leaks involving explicit Intents and components other than Activities. Also, it does not handle some parameters for implicit Intents (such as mimetype and data) and thus generates false links between components. The consequence of that is a higher false positive rate.

SCanDroid [13] and SEFA [14] are two tools that perform ICC analysis. However, neither of them keeps the context between components and thus are less precise than IccTA by design. ComDroid [11] and Epicc [3] are two tools that tackle the ICC problem, but mainly focus on ICC vulnerabilities and do not taint data.

Authors of [15] propose a framework including a set of criteria for evaluating security solutions for smartphones. Their study focuses on the assessment of security solutions, assessing the completeness and the quality of protection capablities of these solutions. Our study analyzes how data leakage is performed on a wide dataset of android malware.

Damopoulos et al. [16] developed an anomaly based intrusion detection system tailored for malware detection and privacy invasive software. The solution is supported by cloud-based technology for exploiting a greater computational power. This work focuses on the solution while our work focuses on the analysis of data leakage in android malware.

Authors in [17] propose a cloud-based smartphone-specific intrusion detection and response engine, which continuously performs an in-depth forensics analysis on the smartphone to detect any misbehavior. The solution is designed specifically for intrusion detection.

Andromaly [18] is a framework for detecting malware on Android mobile devices, through a host-based malware detection system that continuously monitors different events and features. This solution uses a classifier in order to determine whether an application is malicious or not. The focus of this paper is not specifically on data leakage, but malware that can be also steal sensitive data.

3. The Tool Chain

To fully characterize the data leakage phenomenon in Android environment we adopt the tool chain depicted in Figure 1.

The tool chain employed in our analysis is composed by the following open-source software: FlowDroid [1], Amandroid [2] and Epicc [3].

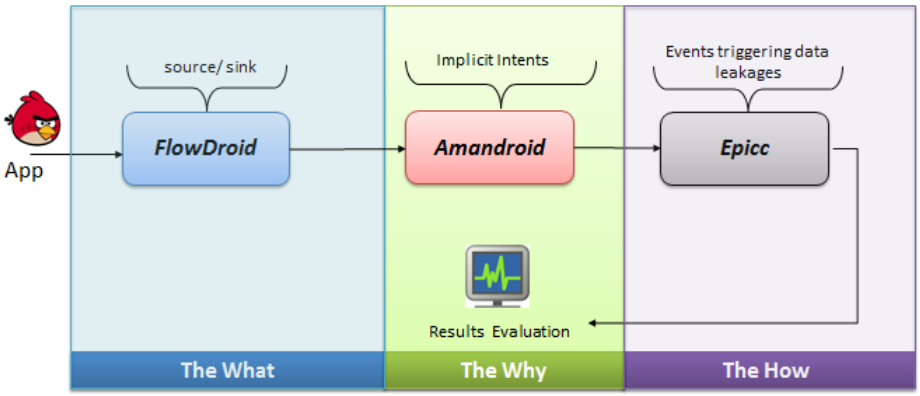

Figure 1: The tool chain bused in the study. We submit the application under analysis to FlowDroid tool, in order to discover the data leakage patterns (the *what*). To understand *why* it is possible that user personal information is sent to third-parties servers, the implicit intents that mobile attackers are able to exploit are intercepted by Amandroid. Finally, by using the Epicc tool, we know *how* the malicious payloads are able to steal personal information.

First of all, we submit each application under analysis to FlowDroid tool in order to understand *what* is the data leakage performed in the device, i.e. we analyse the kind of data that is stolen from the device and the communication channel used to send the data.

To discover the reason *why* the mobile application can send information to a third-party server, we employ Amandroid to deeply discover the implicit intents mechanism used by the application under analysis. Finally, to analyse *how* an application sends the stolen information outside the device, Epicc tool identifies the events that typically activate the data exfiltration in Android environment.

In this section we provide a high-level description of each one of the three tools.

3.1. FlowDroid

FlowDroid is a tool designed for the Android platform able to analyze contaminated paths within the application, i.e. the paths that are able to produce a data exfiltration. This analyzer identifies the full set of the possible connections between the sources and the sinks crossed during a data exfiltration.

In addition, it has the ability to model the full life cycle of an Android application, with particular regards to the management of callback methods and internal interfaces.

Figure 2 shows the source code of an Activity, that is basically the user interface of an Android application. Whenever the Activity is called via the `onRestart()` method, the application reads a password entered using a text field (line 5). When the user clicks on one of the Activity buttons, invoking the callback method `sendMessage()`, the user removes the password and sends it by SMS (line 24). Therefore, this flow between the password entered by the user and the sent SMS, constitutes the data stream contaminated, causing the data loss (the lost data is represented by the password typed by the user). Android developers can

explicitly define the callback method in the Java file of the Activity or implicitly in the XML layout file (see the code snippet in 2): FlowDroid analyzes the source code and processes accurately the associated metadata with the callback methods, i.e. the XML layout files. In addition, the information is actually lost, only if the `onRestart()` method is called before the invocation of the `sendMessage()` method; the analysis performed by FLowDroid is able to understand this order, that is defined flow-sensitive.

```
1  public class LeakageApp extends Activity{
2  private User user = null;
3  protected void onRestart(){
4    EditText usernameText =
       (EditText)findViewById(R.id.username);
5    EditText passwordText =
       (EditText)findViewById(R.id.pwdString);
6    String uname = usernameText.toString();
7    String pwd = passwordText.toString();
8    if(!uname.isEmpty() && !pwd.isEmpty())
9      this.user = new User(uname, pwd);
10 }
11 //Callback method in xml file
12 public void sendMessage(View view){
13   if(user == null) return;
14   Password pwd = user.getpwd();
15   String pwdString = pwd.getPassword();
16   String obfPwd = "";
17   //must track primitives:
18   for(char c : pwdString.toCharArray())
19     obfPwd += c + "_"; //String concat.
20
21   String message = "User: " +
22      user.getName() + " | Pwd: " + obfPwd;
23   SmsManager sms = SmsManager.getDefault();
24   sms.sendTextMessage("+44 020 7321 0905",
25     null, message, null, null);
26 }
```

Figure 2: A Code snippet explaining data leakage in Android.

Unlike Java programs, Android applications do not have the main method. Android applications can contain multiple components, each one characterized by its own life cycle: the information about the component that will be launched when the application is started is provided by the application's Manifest file. For this reason, the static analysis requires more effort, as it has multiple entry points from which to start the analysis. In addition, callback methods have to be considered in the analysis, because they permit to record various information and to interact with the user interface and the Activity. Also the order in which these methods are invoked is important, because it can not be determined a priori (because, as previously explained, an Android application does not have main method). However, the callback methods can only be accessed when the corresponding component is running. To address this problem, FlowDroid bases its analysis on the use of the Interprocedural, Finite, Distributive, Subset framework (IFDS) [19] which emulates the life cycle of the components and the callback methods. The IFDS creates a call-graph to analyse the application components starting from the methods that characterize the life cycle (i.e. `onCreate()`,

`onStop()`) implemented in the respective component classes. This graph is then used to scan the calls to the callback methods and gradually extended to include the discovered callbacks, until it reaches a termination point. Once the dummy main method was built, FlowDroid calculates a callgraph using this method as entry point. This method, although expensive, provides a precise mapping between the components and the callback methods, reducing not only the probability of false positives detection, but also the run-time analysis. This method will take account only of the life cycle of the components and their respective callback methods that can actually occur during the execution and that are defined in the XML configuration file of the application (i.e., the Manifest file).

3.2. Amandroid

Amandroid is able to analyze the flow of data between the platform-specific Android components, by using a flow approach, object and context-sensitive, performing a static analysis of applications. It extends some FlowDroid's features, emphasizing the capability to capture the dependencies between the control and the component data.

Furthermore, in order to manage the control between the components and the flow of data, the tool addresses the security problems that arise from the interactions among multiple components of the same application or between components of different applications through the construction of an inter-procedural graph (ICFG). It is also able to acquire the data flow between the respective components crossed, building an Inter-Component Data Graph (ICDG) used to obtain the DDG, i.e. the graph of the data dependency.

Due to the complexity related to the manipulation of inter-component, a static analyzer needs a model of the Android system to track the invocation of the component lifecycle's methods. The Amandroid model of the Android environment is inspired by FlowDroid [1], which uses a "dummy-Main" method to capture all the possible sequences of the methods' invocations as permitted by Android. The Amandroid model also extends the FlowDroid's model by capturing the control and data dependencies among components.

Amandroid decompresses the apk file of the app to analyse and retrieves the dex file, converting it into an IR format for the subsequent analysis. The *dex2IR* translator is a modification of the original *dexdump* tool shipped with the Android platform tool set; the C++ source of the original *dexdump* is available in the Android build package, while authors modified it so that it can also produce the app representation in the IR format.

The environment model is generated in this way in order to emulate the interaction of the Android system with the application, thus it builds the graph of the data flow between the components of the whole application (IDFG). Finally, it includes the control flow graph that covers all the accessible components. By this way it keeps track of all the created objects (even dynamic ones) that flow at any point of the program from the moment they are created, possibly modified, until their termination point. Amandroid creates an additional graph called *data dependence graph (DDG)*, which suggests explicit information in the flow.

The DDG and the IDFG can be applied to various security analysis: data leak detection, data injection detection, Api's misuse detection, and so on.

The environment model employed by Amandroid extends in many points the FlowDroid one. As a matter of fact Amandroid, rather than using a model of the entire application

(app-level) as FlowDroid, uses an environment model for each component belonging to the the application (component-level).

Unlike the app-level model, the model used by Amandroid is more effective to capture the impact of the Android system on the control of data. The point is that each component has its own model that invokes the callback methods.

To obtain this environment model, Amandroid starts to collect the basic information from the content layout files in the resource folder, to be able to collect the callback methods, and then generates the E_c body containing the methods of the life cycle of the component C; finally, it collects the other callback methods in C , through an incremental reachability analysis, following the approach used by FlowDroid. These operations are carried out before carrying out the analysis of the data flow (IDFG).

3.3. Epicc

Epic aims at detecting the vulnerabilities within Android applications.

This approach is able to provide a high precision considered that it investigates how the components interact by solving the parameters of the ICC calls; more precisely, Epicc is able to retrieve explicit and implicit Intents with the correspondent receiver, in this way it is possible to deduce the possible sender-receiver couples (with the exchanged data) of the intents declared by an Android application.

Unlike FlowDroid and Amandroid, Epicc does not perform a taint analysis, i.e. it does not analyze the flow of data between the components and consequently it is unable to find the paths in which the actual leakage of sensitive data occurs.

3.3.1. The ICC vulnerabilities

Android provides a communication system for exchanging data between the components of the same application, or with different applications, based on the intents. These messages can be sent explicitly by specifying the name of the target component or they may be implicit by specifying the required action: in that case the system will identify the recipient.

Android then determines those intents that must be delivered to the component by matching the name of the target component (in the case of explicit intent) or action, a category and additional data in the case of implicit Intent.

Moreover, several applications can set the same intent-filter and then be candidates to receive the same message; in that case Android will show a window with the candidate applications and the user will choose the application to perform.

In addition, the operating system can also set priorities for all those applications that handle the same type of intent-filter, thus making the Implicit message to be sent to the application with the highest priority.

These types of implicit intent do not offer any guarantee that the message has been delivered to the appropriate recipient, and then malicious app can safely intercept an implicit intent by simply declaring an intent-filter for all the actions, categories, and data of that intent.

3.3.2. The Epicc analysis model

Epicc is focused on the connection between the several components that constitute an Android application, and examines the components communication in the same application and the components communication among multiple applications.

For each input of an application A, Epicc returns the following results:

- a list of the links formed by the A entry-points, that can be called by components of A or from external applications;
- a list of the links formed by the A exit-points, useful to A in order to send Intents to another component of A or to another application;
- a list of the links between the components of A and a list with the links with A and the components of external applications.

To better explain how Epicc works, we consider the code snippet shown in Figure 3 belonging to a banking application.

```
1  public void onClick(View v) {
2    Intent i = new Intent();
3    i.putExtra("Balance", this.mBalance);
4    if (this.mCondition) {
5      i.setClassName("a.b",
           "a.b.MyClass");
6    } else {
7      i.setAction("a.b.ACTION");
8      i.addCategory("a.b.CATEGORY");
9      i = modifyIntent(i);
10   }
11   startActivity(i); }
12
13 public Intent modifyIntent(Intent in) {
14   Intent intent = new Intent(in);
15   intent.setAction("a.b.NEW_ACTION");
16   intent.addCategory("a.b.NEW_CATEGORY");
17   return intent; }
```

Figure 3: An example of Intent-based communication.

In Figure 3 an exit-point is represented by the `startActivity()` method, from which we obtain the result for the i destination and for all the other destinations.

The Epicc analysis comprises the following steps:

1. given an input application, Epicc decompiles it and extracts the Manifest file and information about the packages, the permission, the Intent-Filter and a list of associated components;
2. Epicc performs a matching between the entry-points and the exit-points obtaining the so-called ICC-connections;
3. the obtained objects are stored into a DBMS;

4. it then proceeds with the String Analysis, i.e. the phase through which it is possible to identify, for example, the names of the components or the values of the arguments contained in the various library functions;

5. through the values obtained, the tool invokes the Interprocedural Distributive Environment (IDE) Analysis, which is able to compute the values of Intent used and the methods of the ICC calls. It also computes the values of the Intent-filters that receive Intents through Broadcast Receivers dynamically registered;

6. the exit-points are matched with the entry-points previously computed;

7. the exit-points are stored into the DBMS;

8. the values dynamically associated to the Broadcast Receivers are matched to the exit points;

9. the entry-points are stored into the DBMS.

4. The Experiment

In this section we discuss the result of the experiment we performed to discover the patterns of data leakage in Android malware.

4.1. Dataset

The dataset includes 4593 real world samples gathered from the Drebin project's dataset [20, 21], and 672 samples of ransomware. The Drebin project's dataset [20, 21] is a very well known collection of malware used in many scientific works, which includes the most diffused Android families. Ransomware is a malware that impedes the access to smartphone resources and demands a payment for restoring the functionality and the resources. The ransomware real world samples examined in the experiment were gathered from a freely available collection [1] for research purposes. The samples are labeled as ransomware, koler, locker, fbilocker and scarepackage [22] and appeared from December 2014 to June 2015.

The malware dataset is also partitioned according to the *malware family*: each family contains samples which have in common several characteristics, like payload installation, the kind of attack and the events that trigger malicious payload [23].

We submitted the full dataset to confirm the maliciousness to the VirusTotal service[2]. VirusTotal provides a public API to submit and scan applications, to access finished scan reports of 57 different antimalwares.

Table 1 shows the number of the malware samples with the family they belong to.

[1] `http://ransom.mobi/`

[2] `https://www.virustotal.com`

4.2. Patterns of Data Leakage

In this section we present the results of the evaluation and we discuss the data leakage patterns extracted from the analysed malware.

Table 2 shows the sources retrieved by FlowDroid, i.e. the methods used by the malware to gather information, and provides how many times each method occurs in the samples of each malware family.

We explain in the following the kind of information retrieved by the involved methods:

- `getDeviceId`: it returns the unique device ID, for example, the IMEI for GSM and the MEID or ESN for CDMA phones;
- `getLongitude`: gets the longitude, in degrees;
- `getLatitude`: gets the latitude, in degrees;
- `getCountry`: it returns the country/region code for this locale, which should either be an empty string;
- `getLastKnownLocation`: it returns a Location indicating the data from the last known location fix obtained from the given provider;
- `getSubscriberId`: it returns the unique subscriber ID, for example, the IMSI for a GSM phone;
- `getSimSerialNumber`: it returns the serial number of the SIM, if applicable;
- `getInstalledPackages`: it returns a List of all the packages that are installed on the device;

Table 1: Malware families involved in the experiment with the correspondent number of samples

Family	**# of samples**
FakeInstaller	925
Ransomware	683
DroidKungFu	667
Plankton	625
OpFake	613
GinMaster	339
BaseBridge	330
Kmin	147
Geinimi	92
Adrd	91
DroidDream	81

	Adrd	BaseBridge	DroidDream	DroidKungFu	FakeInstaller	Geinimi	GinMaster	Kmin	Opfake	Plankton	Ransomware
getDeviceId	204			1299	330		11251	340		1299	602
getLongitude	29	32	96	524		92	473			524	1
getLatitude	29	32	96	524		92	473			524	1
getCountry	6	26	24	187		14	91			187	6
getLastKnownLocation	13	9	3	269		71				269	1
getSubscriberId	189	7	4	946	48	72	10691	422	1	946	1
getSimSerialNumber	89				4		10604	72		661	
getInstalledPackages	5	7		98			109			98	
getInstalledApplications	7		36								
getLine1Number	58	2		825	465	2	10615			825	48
getCid		2			1					1	
getLac		2			1					1	

Table 2: Cumulative Occurrences of the Source methods retrieved by FlowDroid on the overall data set ordered by malware family.

- `getInstalledApplications`: it returns a List of all the application packages that are installed on the device;
- `getLine1Number`: it returns the phone number string for line 1, for example, the MSISDN for a GSM phone;
- `getCid`:this method returns the cell tower location. The cell site or cell tower is a cellular telephone site where antennae and electronic communications equipment are placed, usually on a radio mast, tower or other high place, to create a cell (or adjacent cells) in a cellular network;
- `getLac`: it returns the Location Area Code (LAC). The served area of a cellular radio network is usually divided into location areas. Location areas are comprised of one or several radio cells. Each location area is given an unique number within the network, i.e. the LAC. This code is used as a unique reference for the location of a mobile subscriber. This code is necessary to address the subscriber in the case of an incoming call.

As shown in Table 2, all the analysed malware families retrieve the subscriber ID (i.e., the IMSI) using the `getSubscriberId` method. The methods usually employed by malware writers to gather the information about the localization are the `getLongitude` and

	Adrd	BaseBridge	DroidDream	DroidKungFu	FakeInstaller	Geinimi	GinMaster	Kmin	Opfake	Plankton	Ransomware
Log	677	254	216	2410	170	270	4344	647	3	2410	143
HTTP	429	400	347	5134	882	685	5482	1876	1	5134	1056
SharedPreferences	211	1333	53	2716	1404	102	7039	867		2716	3319
File	108	426	93	311	77	3	270	72		311	304
Media	12			2		71					600
SMS	4			12	1387			72	61	12	13

Table 3: The Sink categories retrieved by FlowDroid.

`getLatitude` methods, while the `getCid` and `getLac` methods, that retrieve localization using gms cell location and location area code, are less employed from malware writers, because if compared with GPS precision they are less accurate.

Table 3 shows the sink categories retrieved by FlowDroid, i.e. the channel used to send the gathered information.

We explain in details each sink category retrieved:

- *Log*: this category represents the Android API for creating logs and sending them outside the device. This category can be used to debug the application but also by malware writers to gather information. Error, warning and info logs are always kept. Malware writers are typically interested by the info logs of the application;

- *HTTP*: this category represents the URL Connection with support for HTTP-specific features. It is the preferred method to write personal information into a socket used by malware. The HTTP channel is usually used to communicate with a Command and Control server, to send the personal information to attackers and/or to third-party servers, but also to download at runtime the malicious payload [24]. The classes that are involved in this category are: `DefaultHttpClient`, `BasicNameValuePair`, `URL`, `URLConnection`, `HttpClient`, `OutputStream` and `Write`;

- *SharedPreferences*: Android provides many ways for storing the data of an application. One of this way is *Shared Preferences*. *Shared Preferences* permits to save and retrieve data in the form of (key,value) pair. *Shared Preferences* are stored as a file in the file system of the device. They are, by default, stored within the app's data directory, and only the UID associated to the specific running application have the permissions to access them. The class belonging to this category is `SharedPreferences.Editor` and is invoked by the following methods used to insert information in the SharedPreferences: `putFloat`, `putInt`, `putBoolean`, `putLong`, and `putString`;

- *File*: this category represents the storage on a file. Android uses a file system that is similar to disk-based file systems on other platforms. All the Android devices have two file storage areas: "internal" and "external" storage. These names come from the early days of Android, when most devices offered built-in non-volatile memory (internal storage), plus a removable storage medium such as a micro SD card (external storage). The main difference between internal and external storage is that the second one is world-readable, so files saved here may be read outside of the owner's control, while, by using the internal one, files saved are accessible only by the app itself;

- *Media*: this category is referred to all the media generated by the application, for instance the `onPictureTaken()` method from `Camera.PictureCallback` class, that is called when an image is available after a picture is taken;

- *SMS*: SMSs are used by malware to send messages to premium rate numbers without the user's involvement. Malware also captures the user's banking information such as account number and password [23]. Malware uses also SMSs in order to communicate with C&C server and/or to send SMSs with the malicious links to propagate the infection. The class involved in this category is SMSManager with the invocation of the following methods related to SMS sending: `sendTextMessage`, `sendMultipartTextMessage` and `sendDataMessage`.

Table 4 shows the URLs retrieved by Amandroid with the correspondent number of Implicit Intents.

Table 4: URLs retrieved by Amandroid with the correspondent number of Implicit Intents (#II) and the family they belong to.

Family	**URL**	**#II**
Adrd	http://www.coolcode.org/android/Download_Service.apk	16
	htttp://www.10086apk.com	3
BaseBridge	http://www.androidlicenser.com/store_fronts/3/buy	10
DroidDream	http://pay.sztone.com/czwap/r.aspx	28
	http://market.android.com/search	8
	http://pay.sztone.com/billing/billing.aspx	8
	http://www.opda.com.cn	4
	http://www.kfkx.net/AndroidOptimizer/Weibo2.0.4-2464_0001.apk	4
	http://wp.me/pP0KO-f/	4
	http://market.android.com/details	3
	http://www.google.com.hk/m/search	2
FakeInstaller	http://yandex.ru	97
Geinimi	http://www.dseffects.com/android/games/MonkeyJump2/hi.php	6
	http://www.dseffects.com/android.php	9
GinMaster	http://market.android.com/searh	31
	http://www.netmite.com/android/andme_signed.apk	16
	http://www.amoneron.com/slugs/scoretable.php	11
Kmin	http://www.5j51.com/ThemeDowner/91pandahome2.apk	534

Amandroid found malicious URLs in Adrd, BaseBridge, DroidDream, FakeInstaller, Geinimi, GinMaster and Kmin families. DroidDream is the family that makes an intensive use of malicious URLs with the correspondent implicit intents.

The malicious payload, as explained in [23], can be installed in different ways into a legitimate application:

- *Repackaged*. With repackaging, the malicious payload is embedded into the application at installation time: the attacker decompiles a trusted application to obtain the source code, then adds the malicious payload and recompiles the application;

- *Update attack*. An apparently innocuous application is installed on the victim's device, the user is asked to update the application, which consists of downloading the malicious payload on the victim's device; thus the user has installed an app that does not exhibit any harmful behavior. With this technique, the malicious payload is not embedded into the application at installation time;

- *Drive-by-download*. With this technique the user is asked to download an add-on by clicking on an url or by scanning a QR code: the add-on represents the malicious payload that will be embedded into the legitimate application;

- *Rogueware*. It is a form of malicious software and Internet fraud that misleads the user into believing there is a malware on the device, and manipulates the user into paying money for a fake malware removal tool (that usually actually introduces malware into the computer). It is a form of scareware that manipulates users through fear.

Once installed, the malicious payload is triggered by a set of events, as demonstrated in [23] the most used events able to activate malicious actions are:

- *BOOT*. Most of malware payloads is launched when the boot is completed (BOOT_COMPLETED event), activating a background service that does not require user interaction;

- *SMS*. The SMS_RECEIVED event is transmitted to the system when a new SMS messaged is received. With this event, the malware has the ability to respond to specific incoming SMS messages to undertake malicious actions;

- *NET*. The CONNECTIVITY_CHANGE event is transmitted when a change in the data connection occurs, for instance when the connection switches from GPRS to HSDPA;

- *BATT*. Within this malware feature, we group together a set of events related to battery consumption: ACTION_POWER_CONNECTED (i.e., the device is connected to the power), ACTION_POWER_DISCONNECTED (i.e., the device is disconnected from the power), BATTERY_LOW (i.e., low battery), BATTERY_OKAY (i.e., the battery is now okay after being low), BATTERY_CHANGED_ACTION (broadcast containing the charging state, level, and other information about the battery);

- *SYS*. With this malware feature, we refer to many system events: USER_PRESENT (useful to recognize when the phone has been unlocked or not), INPUT_METHOD_CHANGED (an input method has been changed), SIG_STR (listening to signal strength when the phone sleeps) and SIM_FULL (the SIM storage for SMS messages is full);

- *USB*. The malware is activated when the device is plugged/unplugged, by using the USB cable: it uses the UMS_CONNECTED event in order to know when the device is plugged and the UMS_DISCONNECTED to know when the device is unplugged;

- *PHONE*. The malware responds to the READ_PHONE_STATE event: it allows to access the phone state, including the phone number of the device, the current cellular network information, the status of any ongoing calls, and a list of any phone account registered on the device;

- *PKG*. This category comprises the following events: BROADCAST_PACKAGE_REMOVED, that allows an application to broadcast a notification about the removal of an application package; DELETE_PACKAGES, that allows an application to delete packages; GET_PACKAGE_SIZE, that allows an application to find out the space used by any package; INSTALL_PACKAGES, that allows an application to install packages, PACKAGE_USAGE_STATS, that allows an application to collect usage statistics and REQUEST_INSTALL_PACKAGES, that allows an application to request installing packages;

- *CLOUD*. This category represents the set of Google Cloud Messaging (GCM) events, that is a free service that enables developers to send messages between a server and a client app. This includes downstream messages from a server to a client app, and upstream messages from a client app to a server.

Table 5 shows the events to activate the malicious payload retrieved by Epicc.

Epicc tool, as shown in Table 5, identifies the BOOT events as the events used by all the families in the dataset to trigger the malicious behaviour. Events used by most of malware families are also the SMS event and the NET one. FakeInstaller is the only family that used the GMC events in order to communicate with the attackers and/or third-party servers.

5. Conclusion

In this book chapter we examine the most common code patterns and mechanisms that Android malware is used to adopt to obtain sensitive information from mobile devices. The analysis spans over about 5000 real-world Android malwares, belonging to the most diffused families, including recent ransomware samples.

It emerged that all the families involved in the experiment contain malicious payload that are able to exfiltrate personal information. Ordinarily the information stolen are the IMEI, the phone number, the GPS coordinates, while the most used channels to send the private data are the HTTP connection and the SharedPreferences.

References

[1] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, P. McDaniel, Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps, ACM SIGPLAN Notices 49 (6) (2014) 259–269.

Family	Installation				Activation								
	Repackaging	Update Attack	Drive-by-download	Rogueware	BOOT	SMS	NET	PHONE	USB	PKG	BATT	SYS	CLOUD
Adrd	✓	✓			✓	✓	✓			✓			
BaseBridge	✓	✓			✓	✓	✓		✓		✓	✓	
DroidDream	✓		✓		✓	✓		✓		✓			
DroidKungFu	✓				✓	✓	✓	✓	✓	✓	✓	✓	
FakeInstaller	✓				✓	✓	✓	✓				✓	✓
Geinimi	✓				✓	✓							
GinMaster	✓		✓		✓		✓			✓		✓	
Kmin		✓			✓	✓						✓	
OpFake	✓				✓	✓	✓	✓				✓	
Plankton	✓	✓			✓		✓	✓					
Ransomware		✓	✓	✓	✓	✓	✓	✓				✓	

Table 5: The events to activate the malicious payloads retrieved by Epicc.

[2] F. Wei, S. Roy, X. Ou, et al., Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps, in: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, ACM, 2014, pp. 1329–1341.
[3] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, Y. Le Traon, Effective inter-component communication mapping in android: An essential step towards holistic security analysis, in: Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13), 2013, pp. 543–558.
[4] J. Kim, Y. Yoon, K. Yi, J. Shin, S. Center, Scandal: Static analyzer for detecting privacy leaks in android applications, MoST 12.
[5] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, A. N. Sheth, Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones, ACM Transactions on Computer Systems (TOCS) 32 (2) (2014) 5.
[6] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, P. McDaniel, Iccta: Detecting inter-component privacy leaks in android apps, in: Proceedings of the 37th International Conference on Software Engineering-Volume 1, IEEE Press, 2015, pp. 280–291.
[7] L. Lu, Z. Li, Z. Wu, W. Lee, G. Jiang, Chex: statically vetting android apps for component hijacking vulnerabilities, in: Proceedings of the 2012 ACM conference on Computer and communications security, ACM, 2012, pp. 229–240.
[8] L. Li, A. Bartel, J. Klein, Y. Le Traon, Automatically exploiting potential component leaks in android applications, in: 2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications, IEEE, 2014, pp. 388–397.
[9] Y. Z. X. Jiang, Detecting passive content leaks and pollution in android applications, in: Proceedings of the 20th Network and Distributed System Security Symposium (NDSS), 2013.
[10] D. Papamartzivanos, D. Damopoulos, G. Kambourakis, A cloud-based architecture to crowdsource mobile app privacy leaks, in: Proceedings of the 18th Panhellenic Conference on Informatics, PCI '14, ACM, New York, NY, USA, 2014, pp. 59:1–59:6. doi:10.1145/2645791.2645799.
URL http://doi.acm.org/10.1145/2645791.2645799
[11] E. Chin, A. P. Felt, K. Greenwood, D. Wagner, Analyzing inter-application communication in android, in: Proceedings of the 9th international conference on Mobile systems, applications, and services, ACM, 2011, pp. 239–252.
[12] W. Klieber, L. Flynn, A. Bhosale, L. Jia, L. Bauer, Android taint flow analysis for app sets, in: Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis, ACM, 2014, pp. 1–6.
[13] A. P. Fuchs, A. Chaudhuri, J. S. Foster, Scandroid: Automated security certification of android.
[14] L. Wu, M. Grace, Y. Zhou, C. Wu, X. Jiang, The impact of vendor customizations on android security, in: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, ACM, 2013, pp. 623–634.
[15] A. Shabtai, L. Tenenboim-Chekina, D. Mimran, L. Rokach, B. Shapira, Y. Elovici, Mobile malware detection through analysis of deviations in application network behavior, Computers Security 43 (2014) 1 – 18. doi:http://dx.doi.org/10.1016/j.cose.2014.02.009.
URL http://www.sciencedirect.com/science/article/pii/S0167404814000285
[16] D. Damopoulos, G. Kambourakis, G. Portokalidis, The best of both worlds: A framework for the synergistic operation of host and cloud anomaly-based ids for smartphones, in: Proceedings of the Seventh European Workshop on System Security, EuroSec '14, ACM, New York, NY, USA, 2014, pp. 6:1–6:6. doi:10.1145/2592791.2592797.
URL http://doi.acm.org/10.1145/2592791.2592797
[17] A. Houmansadr, S. A. Zonouz, R. Berthier, A cloud-based intrusion detection and response system for mobile phones, in: Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops, DSNW '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 31–32. doi:10.1109/DSNW.2011.5958860.
URL http://dx.doi.org/10.1109/DSNW.2011.5958860
[18] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, Y. Weiss, "andromaly": A behavioral malware detection

framework for android devices, J. Intell. Inf. Syst. 38 (1) (2012) 161–190. doi:10.1007/s10844-010-0148-x.
URL http://dx.doi.org/10.1007/s10844-010-0148-x

[19] T. Reps, S. Horwitz, M. Sagiv, Precise interprocedural dataflow analysis via graph reachability, in: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, 1995, pp. 49–61.

[20] D. Arp, M. Spreitzenbarth, M. Huebner, H. Gascon, K. Rieck, Drebin: Efficient and explainable detection of android malware in your pocket, in: Proceedings of 21th Annual Network and Distributed System Security Symposium (NDSS), IEEE, 2014.

[21] M. Spreitzenbarth, F. Echtler, T. Schreck, F. C. Freling, J. Hoffmann, Mobilesandbox: Looking deeper into android applications, in: 28th International ACM Symposium on Applied Computing (SAC), ACM, 2013.

[22] N. Andronio, S. Zanero, F. Maggi, Heldroid: dissecting and detecting mobile ransomware, in: International Workshop on Recent Advances in Intrusion Detection, Springer, 2015, pp. 382–404.

[23] Y. Zhou, X. Jiang, Dissecting android malware: Characterization and evolution, in: 2012 IEEE Symposium on Security and Privacy, IEEE, 2012, pp. 95–109.

[24] F. Mercaldo, V. Nardone, A. Santone, C. A. Visaggio, Download malware? No, thanks. How formal methods can block update attacks, in: Formal Methods in Software Engineering (FormaliSE), 2016 IEEE/ACM 4th FME Workshop on, IEEE, 2016.