Journal of Engineering and Applied Sciences 12 (Special Issue 2): 6308-6312, 2017
ISSN: 1816-949X


# TaintDroid's Ability Test and Remedy for Tainting SMS, Bookmark, Accelerometer and Call Log Information

Han-Jae Yoon and Man-Hee Lee
Department of Computer Engineering, Hannam University, 70, Hannam-ro, Daedeok-gu, Deajoen, Republic of Korea

**Abstract:** As Android malware continues to grow, automated analysis using the virtual environment is very necessary. TaintDroid is an efficient tool that allows tainting analysis to monitor whether key information is leaked to the outside. TaintDroid is expected to detect apps that leak SMS, phone logs and contact information but in our tests it failed to do so. In this study, we explained the reason and proposed a simple solution.

**Key words:** Malware Android, tainting analysis, TaintDroid, simple, reason, SMS

## INTRODUCTION

As the number of mobile devices increases rapidly, malicious codes become more interested in stealing business and personal information stored in the devices. A common process to cope with malicious apps will be collecting apps, analyzing them statically or dynamically and generating signatures for vaccines. Throughout the process, analyzing the maliciousness of app is the most important and difficult. Static analysis on source codes or packaged Android Application Packages (APKs) looks at various static aspects such as permissions, strings and Android manifest. In dynamic analysis, an Android application runs on a virtual or physical system and its behaviors like file and network accesses are monitored and analyzed.

Among various dynamic analysis tools, TaintDroid is famous for its ability of tracking interested information throughout execution which is called tainting analysis. In particular, TaintDroid has predetermined data for tainting. If the analyzed application writes this data to a file or leaks it to the internet, a data leak event occurs and a log about the event is stored. The strength of TaintDroid is that it can detect the leakage of even one bit tainted data. This enables TaintDroid to detect any malicious codes that leak confidential or private data after encoding, slicing or encrypting.

In our previous study, we found that TaintDroid was unable to detect a malicious code that leaked contact information. We figured out that a clean Android system does not have contact information at all and this caused the detection failure of the malicious code. In addition, we proposed a method to solve the problem by inserting contact information before the start of analysis. This research expanded the previous study by exploring other tainting sources. That is, we tested more tainting information such as SMS, bookmark, call log and accelerometer. TaintDroid is able to detect malware leaking bookmark and accelerometer information but not able to do on malware leaking SMS and call log for the same reason. We also found a simple bug of TaintDroid for tainting call log information and fixed it. This study elaborated on our tests and solutions.

**Literature review:** Current CPU architectures do not provide detailed information on how data is processed. For example, it is possible to monitor what data in a file is read but not possible to see how the data is changed and stored in memory which file it is stored in and which hosts it is transferred to. This means it is very difficult to perform tainting analysis on real computers. To make this possible in virtual Android system, TaintDroid was proposed (Enck *et al*., 2010). It is almost the first practical tainting analysis tools for Android (Enck *et al*., 2010). It is developed to expand the Android platform. The main difference from previous techniques is that TaintDroid utilizes virtual machine-based architecture of Android. Similar to Java virtual machine running byte codes for normal Java applications, Dalvik VM Interpreter runs Dalvik EXecutable (DEX) byte codes for Android applications. Each Android application operates within an instance of Dalvik VM interpreter. By modifying the interpreter, TaintDroid tracks variable level information tracking. In addition to variable level information tracking, TaintDroid suggested method-level, file-level and message-level tracking. Method-level tracking is

**Corresponding Author:** Han-Jae Yoon, Department of Computer Engineering, Hannam University, 70, Hannam-ro, Daedeok-gu, Deajoen, Republic of Korea

implemented in native system library which is out of VM interpreter. Network interface and secondary storage are also out of VM interpreter, so, TaintDroid provides a patch for information tracking. Finally, TaintDroid tracks data transferred between VM instances via. IPC (interprocess communications). By using the four levels of tracking, TaintDroid successfully tracks information with marginal performance overhead.

By utilizing tracking capability, TaintDroid monitors various information to see when and how the data is written to a file or sent via. a network interface. In addition, TaintDroid with DroidBox as a dynamic behavior analysis tool gives out the following information: incoming/ outgoing network data, file read/write operation, started services and so on (Enck *et al*., 2010).

Since, TaintDroid was proposed, many researchers tried to utilize it for better Android malware analysis. Especially, static analysis tools are integrated with TaintDroid to provide both dynamic and static analysis by Lindorfer *et al*. (2014), Yuan *et al*. (2014) and Spreitzenbarth *et al*. (2013). In the separate previous study, we constructed an automatic app analysis system utilizing TaintDroid to investigate how it is to use apps downloaded from 3rd-party app stores (Jang *et al*., 2016). There have been other researchers to enhance detectability of TaintDroid. Many apps require user interactions in the installation process. Otherwise, malware will not be installed, so malicious behaviors cannot be detected, either. To solve this problem, several techniques generating user interface events to emulate user interactions have been proposed by Rastogi *et al*. (2013), Zheng *et al*. (2014), Michelle and Lie (2016) and Hao *et al*. (2014). However, to our best knowledge, we do not know any research to test the tainting ability of TaintDroid except our previous research by Yoon and Lee (2017).

## MATERIALS AND METHODS

**Detection failure of TaintDroid:** In previous study, we found that a clean Android image that TaintDroid uses does not have contact information (Yoon and Lee, 2017). Because of this, malware that leaks contact information cannot be detected. As a real example, we tested a malware called Alsalah. It was advertised that it informs five Salah (prayer in Islam) timings but in fact it is a Trojan horse app that sends spam SMS messages to contacts in the compromised device. With no contact information created, TaintDroid does not detect anything suspicious on Alsalah. So, we proposed to modify Android system to have at least one contact information.

The next question that we are curious about is whether the detection failure problem can occur in other tainting sources. So, in this study, we would like to expand our previous research by testing if TaintDroid detects malware leaking other tainting sources. Instead of testing all the remaining sources, we chose four additional tainting sources: SMS, bookmark, call log and accelerometer. We expected SMS, call log and browser bookmark information would not be detected because a clean Android image might not have such information. Instead, malware leaking acceleration data from an accelerometer is predicted to be detected because the accelerometer would keep generating new data.

For test, we developed a sample malware that accesses and write the data to a file. When we ran the sample app on TaintDroid, it reported data leakage of the accelerometer and the browser bookmark information only as shown in Algorithm 1. It is reasonable to see a Taint_Accelerometer event but the Taint_Browser event was unexpected because we assumed that there was no bookmark information. Base on the test result, we can say that it is not straightforward to predict whether TaintDroid detects all the advertised sink sources and it is not clear either why this happens. In the following study, we continue to answer why and how TaintDroid is able or unable to detect some sink sources and propose how to fix the problem.

**Algorithm 1; Original Taint analysis result of TaintDroid for SMS, bookmark, call log and accelerometer:**


```
"dataleaks": {
    "2.1697490215301514": {
      "Method": "None"
      "Package": "None"
       "data":
"3d3d3d3d3d2043616c6c204c6f67203d3d3d3d3d0a6e756c6c0a3d3d3d3d
3d3d3d3d3d3d3d3d3d3d3d3d3d3d0a3d3d3d3d3d20486973746f7279203d3d
3d3d3d0a6e756c6c426f6f6b6d61726b730a476f6f676c650a5069636173610
a5961686f6f210a4d"
      "id": "1238985775"
      "operation": "write"
      "path": "/mnt/sdcard/taintTest.txt"
      "sink": "File"
      "tag": [
"TAINT_ACCELEROMETER"
        "TAINT_BROWSER"
      ]
      "type": "file write"
    }
}
```


## RESULTS AND DISCUSSION

**Tainting enalbing techniques for TaintDroid:** This study explains the reason why TaintDroid succeeded or failed to detect the browser bookmark, call log and SMS and effective tainting enabling techniques.

**Browser bookmark:** In order to see why TaintDroid was able to detect bookmark information leaking apps, we developed an app to extract bookmark information from a clean Android image as shown in Algorithm 2. We implemented a method, get History(), to get the bookmark information using Cursor class and BOOKMARKS_URI. When we checked mCur, it contains 14 bookmarks: Google, Picasa, Yahoo!, MSN, Twitter, Facebook, Wikipedia, eBay, CNN, NY Times, ESPN, Amazon, Weather Channel and BBC. Based on this test, we can say any malware that extracts bookmark information will be detected by TaintDroid.

**Algorithm 2; Source code for extracting bookmark information:**

```
public String getHistory() {
    String return_str = null
      Cursor mCur = this.managedQuery(BOOKMARKS_URI,
HISTORY_PROJECTION, null, null, null)
    if (mCur.moveToFirst()) {
      while (mCur.isAfterLast() == false) {
    return_str += m Cur. getString (HISTORY_
PROJECTION_ TITLE_INDEX)+"\n"
        mCur.moveToNext()
      }
    } else {
      return_str = "No data"
    }
    return return_str
  }
```

**Short message service:** As we already noted early, TaintDroid did not detect our sample malware leaking SMS. Based on our previous research, we expect this was caused by no initial SMS stored in Android image. By using the same approach we used before, we implemented a method, putSMS() in TelephonyProvider.java shown in Algorithm 3 to insert an SMS having a message, "Injected Message", from a phone number, 01036165366. By using content values class, we set the phone number at address column and the message at body column. Finally, we inserted the content values instance at content://sms/ inbox. After recompilation of Android system, we can check that an SMS message with "Injected Message" from 01036165366 was generated as shown in Fig. 1.

**Algorithm 3; Source code of TelephonyProvider.java for inserting SMS:**

```
public void putSMS() {
  ContentValues values = new ContentValues()
  values.put("address", "01036165366")
  values.put("body", "Injected Message")
getContext().getContentResolver().insert(Uri.parse("content://sms/inbox"),
values)
}
```

**Call history:** The sample malware extracting call history shown in Algorithm 4 was not detected by TaintDroid.

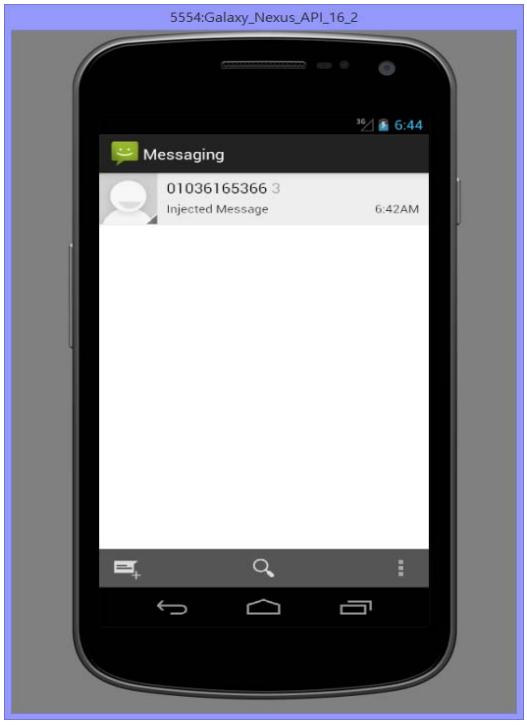


Fig. 1: Inserted SMS


Fig. 2: Inserted call log

We took the same approach as used in the SMS injection to insert a call log having a 60 sec-long incoming call from a phone number, 01036165366, shown in Fig. 2.

Differences compared to SMS injection are injected data and URI and we found the call log was inserted well as shown in Algorithm 5. However, even after inserting call logs, TaintDroid was still unable to detect the malware.

**Algorithm 4; Call history extracting code:**

```
public String getCallLog() {
String return_str = null
ContentValues c = new ContentValues()
try {
StringBuffer sb = new StringBuffer();
Cursor managed Cursor = managedQuery (android.provider.CallLog.Calls.
CONTENT_URI,null, null,null, null)
int number = managedCursor.getColumnIndex (android.provider.CallLog.
Calls.NUMBER)
int type = managedCursor.getColumnIndex ( android.provider.CallLog.
Calls.TYPE)
int duration = managedCursor.getColumnIndex ( android.provider.CallLog.
Calls.DURATION)
sb.append ("Call Details")
while ( managedCursor.moveToNext() ) {
String phNumber = managedCursor.getString(number)
String callType = managedCursor.getString(type)
String callDuration = managedCursor.get
String(duration)
String dir = null
int dircode = Integer.parseInt(callType)
if (callType.equals("1")) {
callType = "incoming"
} else {
callType = "outcoming"
}
return_str = "Num:"+phNumber+"duration:"+ callDuration+"type:"+
callType
}
} catch (Exception e) {
e.printStackTrace();
return_str = "No data";
}
return return_str;
}
```

**Algorithm 5; Source code of TelephonyProvider.java for inserting call history:**

```
Public void putCallLog() {
ContentValues values = new ContentValues()
values.put(android.provider.CallLog.Calls.CACHED_NUMBER_TYPE,
0)
values.put(android.provider.CallLog.Calls.TYPE android. provider.
CallLog.Calls.INCOMING_TYPE)
values.put(android.provider.CallLog.Calls.DATE,System.current
TimeMillis())
values.put(android.provider.CallLog.Calls.DURATION, 60)
values.put(android.provider.CallLog.Calls.NUMBER, "01036165366")
try {
getContext().getContentResolver().insert(android.provider.CallLog.Calls.
CONTENT_URI, values)
} catch (Exception e) {
e.printStackTrace()
  }
}
```

So, we tried to find what caused this symptom and figured out a bug in TaintDroid. ContentResolver.java is originally included in Android system for providing applications access to the content model. TaintDroid modified it as a patch for putting a tag according to data sources. As shown in Algorithm 6, we found that TaintDroid programmers forgot to add codes for checking the call log source, " content://call_log/calls", so TAINT_CALL_LOG was not added appropriately. After modifying the TAINT_CALL_LOG part and building TaintDroid again, we got correct taint analysis results that include four types of tainting tags as shown in Algorithm 7.

**Algorithm 6; Modified source code of ContentResolver.java for tagging call log information:**

```
//begin WITH_TAINT_TRACKING
int taint = Taint.TAINT_CLEAR
if(uri.toString().indexOf("com.android.contacts") != -1) {
  taint = Taint.TAINT_CONTACTS
}
else if(uri.toString().indexOf("browser/bookmarks") != -1) {
  taint = Taint.TAINT_HISTORY
}
else if(uri.toString().indexOf("content://sms") != -1) {
  taint = Taint.TAINT_SMS
}
else if(uri.toString().indexOf("content://mms") != -1) {
  taint = Taint.TAINT_SMS
}
//Added in this research
else if(uri.toString().indexOf("content://call_log/calls") != -1) {
  taint = Taint.TAINT_CALL_LOG
}
// end WITH_TAINT_TRACKING
```

**Algorithm 7; Taint analysis result of modified TaintDroid for SMS, bookmark, call log and accelerometer:**

```
"dataleaks": {
"2.3443870544433594": {
   "Method": "None"
   "Package": "None"
"data": "3d3d3d3d3d20416363656c65726f6d65 746572203d3d3d3d3d0
a58203a20302e30202f2059203a20392e3737363232202f205a203a20302e3
831333431370a3d3d3d3d3d3d3d3d3d3d3d3d3d3d3d3d3d3d0a3d3d3d3d3d
20534d53203d3d3d3d3d0a416464"
   "id": "778061256"
   "operation": "write"
   "path": "/mnt/sdcard/ taintTest.txt"
   "sink": "File"
   "tag": [
    "AINT_ACCELEROMETER"
    "TAINT_SMS"
    "TAINT_CALL_LOG"
    "TAINT_BROWSER"
     ]
     "type": "file write"
  }
}
```

## CONCLUSION

TaintDroid is a very useful tool to run Android applications for tainting analysis. Various data sources are tracked in TaintDroid but we found that some of them

have not been tracked properly due to no initial data or a simple programming bug. In this study, we checked four data sources accelerometer, bookmark, SMS and call log. TaintDroid keeps tagging data from the accelerometer, so its data is successfully tracked. Android system has a list of bookmark, so, malicious applications leaking the bookmark will be detected. However, a clean Android system does not contain SMS data, so, SMS data leaking application cannot be detected. We solved this problem by inserting a sample SMS before start of analysis. Finally, since, there is no call log information in the clean Android system, we inserted a sample log as we did for SMS. However, in call log case, we found a simple bug that TaintDroid missed to attach a tag to call log data, thus resulting in failure of call log leaking applications even after inserting call log information. By modifying a TaintDroid source code, we got correct tainting results.

Currently, we are investigating to see if there are similar detection failures in other remaining tainting sources. In the preliminary tests, some sources like TAINT_EMAIL, TAINT_MIC and TAINT_CAMERA produce tainting failures, too. We expect that the complete information about which tainting sources work or not in clean Android emulator will be useful to other researchers.

## REFERENCES


Enck, W., P. Gilbert, B.G. Chun, L.P. Cox, J. Jung, P. McDaniel and A.N. Sheth, 2010. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation, Volume 10, October 4-6, 2010, Vancouver, BC., Canada, pp: 255-270.

Hao, S., B. Liu, S. Nath, W.G. Halfond and R. Govindan, 2014. Puma: Programmable UI-automation for large-scale dynamic analysis of mobile apps. Proceedings of the 12th Annual International Conference on Mobile Systems, Applications and Services, June 16-19, 2014, ACM, Bretton Woods, New Hampshire, USA., ISBN:978-1-4503-2793-0, pp: 204-217.

Jang, B., J. Lee and M. Lee, 2016. Automatic system for measuring security risk of Android application from third party app store. Secur. Commun. Netw., 9: 3190-3196.

Lindorfer, M., M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio and V.V.D. Veen *et al*., 2014. Andrubis-1,000,000 apps later: A view on current Android malware behaviors. Proceedings of the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS'14), September, 11, 2014, IEEE, Wroclaw, Poland, ISBN:978-1-4799-8308-7, pp: 3-17.

Michelle, Y.W. and D. Lie, 2016. IntelliDroid: A targeted input generator for the dynamic analysis of android malware. Proceedings of the 2016 Symposium on Network and Distributed System Security (NDSS'16), February 21-24, 2016, Internet Society, San Diego, California, pp: 1-15.

Rastogi, V., Y. Chen and W. Enck, 2013. Apps playground: Automatic security analysis of smartphone applications. Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy, February 18-20, 2013, ACM, San Antonio, Texas, ISBN:978-1-4503-1890-7, pp: 209-220.

Spreitzenbarth, M., F. Freiling, F. Echtler, T. Schreck and J. Hoffmann, 2013. Mobile-sandbox: Having a deeper look into android applications. Proceedings of the 28th Annual ACM Symposium on Applied Computing, March 18-22, 2013, ACM, Coimbra, Portugal, ISBN:978-1-4503-1656-9, pp: 1808-1815.

Yoon, H.J. and M.H. Lee, 2017. A tip for enabling taint analysis of contact information in TaintDroid. Proceedings of the 2017 International Conference on Inventive Communication and Computational Technologies (ICICCT'17), March 10-11, 2017, Hannam University, Daejeon, South Korea, pp: 1-3.

Yuan, Z., Y. Lu, Z. Wang and Y. Xue, 2014. Droid-Sec: Deep learning in android malware detection. Proceedings of the ACM SIGCOMM Computer Communication Review Vol. 44, August 17-22, 2014, ACM, Chicago, Illinois, ISBN:978-1-4503-2836-4, pp: 371-372.

Zheng, M., M. Sun and J.C. Lui, 2014. DroidTrace: A ptrace based Android dynamic analysis system with forward execution capability. Proceedings of the 2014 International Conference on Wireless Communications and Mobile Computing (IWCMC'14), August 4-8, 2014, IEEE, Nicosia, Cyprus, ISBN:978-1-4799-0959-9, pp: 128-133.