A Tip for Enabling Taint Analysis of Contact Information in TaintDroid

Han-Jae Yoon

Hannam Univ. Computer Engineering
70, Hannam-ro, Daedeok-gu
Deajoen, Republic of Korea
hanjae.karoha@gmail.com

Man-Hee Lee

Hannam Univ. Computer Engineering
70, Hannam-ro, Daedeok-gu, Hannam Univ.
Deajoen, Republic of Korea
manheelee@hnu.ac.kr

Abstract

As Android malware keeps increasing, automatic analysis using virtual environments becomes very necessary. Among them, TaintDroid is famous for its tainting analysis functionality. Before running malware under test, concerned information is tainted (or marked), and TaintDroid keeps tracking the tainted data wherever it moves around. When it leaves Android system, an alert is generated for further analysis. While we analyzed a malicious app that leaks contact information, we found TaintDroid could not detect the app. In this paper, we presented what caused the situation and proposed a tip to make it possible.

Keywords-component; malware, Android, Tainting Analysis, TaintDroid

I. Introduction

The malware trend is rapidly changing from desktop-based to smartphone-based. According to [1][2], cyber threats on smartphone are becoming very serious.

Automatic analysis of malicious app is essential to cope with this problem. Automatic analysis tools are categorized into static or dynamic. Static analysis is done to source codes or packaged Android application package (APK) to look for various aspects of the app such as permission, strings, Android Manifest, so on[3]. In dynamic analysis, Android application under test is actually run on virtual or real systems. Then, its behavior is monitored for investigating into security related issues.

DroidBox with TaintDroid is a famous dynamic analysis tool for its ability of tainting analysis [4][5]. To taint some data means that a flag is marked on the data. Whenever data assignments occur, its flag was copied to the newly assigned data. It is believed that the tainting analysis would be the most powerful anti-data leak technique.

However, we found an interesting malicious app called contactleak. The contactleak is known to leak contact information. So we expected that it will be simply detected by TaintDroid, but to our surprise TaintDroid does not report the data leakage of contact information. We investigated into this issue and figured out what happened. The contactleak first checked whether there is contact information in an Android system that the app is installed. If there is nothing to leak, it cannot send data. Since a clean Android image kept be used at every malware analysis, there was no contact information on the image. This is why TaintDroid could not detect the contactleak.

In this paper, we showed this symptom and provided a simple tip to detect the contactleak. It is to insert contact data before running the contactleak. To do so, we developed an Android app to do so and run it before contactleak. After this remedy, the contactleak was detected well. Although this finding and the proposed tip may look very simple, our research would be very helpful to other Android malware analysts.

II. Detection Failure of Contact Information Leakage App

A. Malware analysis: AlSalah

Figure 1 shows a snapshot of AlSalah. Its advertised function is to inform users five Salah (prayer in Islam) timings. According to Symantec [6], AlSalah a Trojan horse for Android devices that sends spam SMS messages to contacts on the compromised device. It also gathers the contacts on the compromised device and send each to predefined lists of URLs.

Figure 1. snapshot of AlSalah

B. Detection Failure of TaintDroid

Since TaintDroid sets contact information as private data, it tracks contact information. So we expected TaintDroid would detect it as TAINT_CONTACT, but to our surprise it did not do so. We investigated this issue and found out that AlSalah looked for contact information but could not find anything because a new TaintDroid image does not carry any contact information.

III. Enabling Contact Information Leakage App Detection

A. Approaches for Fixing:

To fix this problem, we considered two approaches. First, we could program a simple app to insert a contact into the list. The other is to modify TaintDroid to have at least one contact information. Based on complexity of implementation, the first approach is more preferable. But we chose to modify TaintDroid for two reasons.

The first reason is the performance problem. When the first approach is applied, the contact inserting app needs to be installed and run prior to installation of AlSalah. We created an app for test and experimented on TaintDroid ten times. It takes 29.41 seconds on average. The time may look short to some people, but it is huge time waste because installing and running malware under test takes the similar amount of time. That is, the total analysis time would be doubled. Considering that virtual environment is commonly used for testing a huge number of malware, such performance overhead is too big.

The second reason is the test complexity. Please imagine that every TaintDroid user needs to run the contact insertion app before testing each app. That would be almost impossible scenario. This is why we decided to modify TaintDroid. If users take our approach, they simply modify TaintDroid once and keep using the modified image.

B. Modification of TaintDroid

We decided to modify *TelephoyProvider* class [7]. When the Android system is booted up, it makes an access point name (APN) that is a gateway between a mobile network and the Internet [8]. Internet accessibility is essential for smart phones so the class's activity should be run at least once. Because of this, we chose the *TelephoyProvider* class as a target modification class.

onCreate() method in TelephonyProvider.java shown in Figure 2 is to initialize activity of the class. The newly created source codes are composed of four parts. The first part is to prepare a container, *ops*, for a contact information. The second is to write a phone number, "7894156", to the container. Then, we write a contact name, "Mr.FAKE", to the containter. Finally, we write the container's information to the system's contact information by calling *getContext().getContentResolver().applyBatch(ContactsContract.AUTHORITY, ops)*. Figure 3 shows a screen capture of the TaintDroid when we check the contact information.

```
public class TelephonyProvider extends ContentProvider
{
  public boolean onCreate() {
    ArrayList<ContentProviderOperation> ops =
              new  ArrayList<ContentProviderOperation>();
    int rawContactInsertIndex = ops.size();

    //preparations
ops.add(ContentProviderOperation.newInsert(RawContacts
.CONTENT_URI)
       .withValue(RawContacts.ACCOUNT_TYPE, null)
       .withValue(RawContacts.ACCOUNT_NAME,
                                              null).build());

    //Phone Number
    ops.add(ContentProviderOperation
       .newInsert(ContactsContract.Data.CONTENT_URI)
       .withValueBackReference(ContactsContract.
                                  Data.RAW_CONTACT_ID,
          rawContactInsertIndex)
       .withValue(Data.MIMETYPE, android.provider.
                     ContactsContract.CommonDataKinds.
                     Phone.CONTENT_ITEM_TYPE)
       .withValue(android.provider.ContactsContract.
           CommonDataKinds.Phone.NUMBER, "7894156")
       .withValue(Data.MIMETYPE, android.provider.
                     ContactsContract.CommonDataKinds.
                     Phone.CONTENT_ITEM_TYPE).
                     withValue(android.provider.
                     ContactsContract.CommonDataKinds.
                     Phone.TYPE, "1").build());

    //Contact Name
    ops.add(ContentProviderOperation
       .newInsert(ContactsContract.Data.CONTENT_URI)
       .withValueBackReference(Data.RAW_CONTACT_ID,
                                     rawContactInsertIndex)
       .withValue(Data.MIMETYPE, StructuredName.
                                    CONTENT_ITEM_TYPE)
       .withValue(StructuredName.DISPLAY_NAME,
                                    "Mr.FAKE").build());

    try {
      ContentProviderResult[] res =
               getContext().getContentResolver().
               applyBatch(ContactsContract.
                                        AUTHORITY, ops);
    } catch (RemoteException e) {
      e.printStackTrace();
    } catch (OperationApplicationException e) {
      e.printStackTrace();
    }
    Taint.log("inject success");
    return true;
  }
  ...omission...
}
```

Figure 2. Edited TelephonyProvicer.java

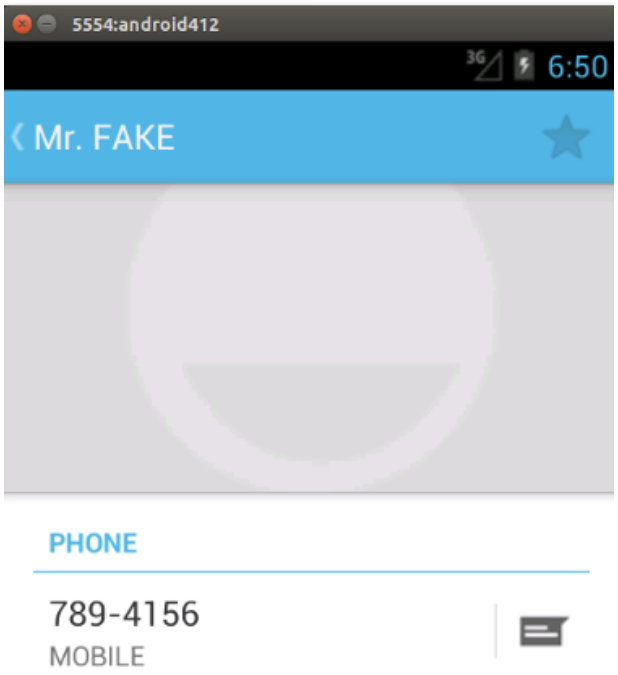

Figure 3. Result of contact information inject

IV. Experiments

With the newly modified TaintDroid, we tested AlSalah again. Figure 4 shows the dataleaks part of the resulting json file generated by TaintDroid. Different from the result when using the original TaintDroid, the dataleaks part provides information about a leaking event. We can understand that a file, "contacts.en_US.dict.temp", was written to a certain path, "/data/data/com.android.inputmethod.latin/files", and the file contains contact information with the TAINT_CONTACTS tag.

Please note that the data leak events are different from what we expected. As Symantec reported, this app is supposed to send contact information to a list of web sites, but we could not find such information. Its reason is that all the web sites were already closed so the app could not send out any packets to the sites. If there were any sites available, we could have found other data leak events, too.


```
"dataleaks": {
      "3.6312878131866455": {
         "Method": "None",
         "Package": "None",
         "data": "023046414b451f28344d721f28530d",
         "id": "962167312",
         "operation": "write",
         "path": /data/data/com.android.inputmethod.latin/files
/contacts.en_US.dict.temp",
         "sink": "File",
         "tag": [
            "TAINT_CONTACTS"
         ],
         "type": "file write"
      }
   },
```


Figure 4. Analysis result

V. Conclusion

In this paper, we looked for a TaintDroid's detection failure case that TaintDroid could not detect a data leak malware, AlSalah. We found it is caused because there is no contact information in the clean TaintDroid image. To solve this problem, we decided to modify TaintDroid for better performance and testability. Among many classes, we chose TelephonyProvider class because its activity is run at least once for Android system's internetworking. After inserting a contact list, TaintDroid was able to detect AlSalah successfully. We hope this tip for enabling TaintDroid's data leak detection for contact information will be helpful to other users.

References

[1] Pew Research Center, "Smartphone Ownership and Internet Usage Continues to Climb in Emerging Economies", 2016.2.J. Clerk Maxwell, *A Treaties on Electricity and Magnetism*, 3rd ed., Vol. 2, Oxford: Clarendon Press, 1892, pp. 68-73.

[2] Intel Security "McAfee Labs Threats Report", 2015.8.

[3] https://android-arsenal.com/tag/94

[4] 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", TOCS, Volume 32 Issue 2, June 2014, Article No. 5.

[5] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, Johannes Hoffmann, "Mobile-sandbox: baving a deeper look into android applications", SAC13, pages 1808-1815

[6] Symantec https://www.symantec.com/security_response/writeup.jsp?docid=2011-121915-3251-99

[7] https://android.googlesource.com/platform/packages/providers/TelephonyProvider/+/4167fcc/src/com/android/providers/telephony/TelephonyProvider.java

[8] Wikipedia https://en.wikipedia.org/wiki/Access_Point_Name