Malicious Activity by Colluding Android Applications

Mark V. Lese
College of Engineering, Technology, and Computer Science
Indiana University - Purdue University Fort Wayne
Fort Wayne, IN 46805
Email: lesemv01@students.ipfw.edu

***Abstract*—This paper provides a technical report for CS59000-09; advised by Dr. Karim Elish. Colluding Android applications pose a security risk to unsuspecting end users. These applications may maliciously spread private personal data the end user thought was secure. This paper describes the overview of colluding applications and the simplicity creating these types of applications. Scanners exist that attempt to detect malicious colluding applications. It is likely, at least some scanners would find the applications discussed in this paper have malicious intent. A technique stumbled upon may thwart such scanners and is left for future work.**

I. Introduction

Android application framework provides the means for applications to communicate with each other using inter-component communication (ICC). While this framework allows for rich application development, the ICC-based communications also facilitates attackers who wish to develop malicious colluding applications using the same techniques used for benign applications[4]. Collusion is the process where two or more applications, written by the same developer (me in this case), communicate with each other providing a virtual set of permissions that allow malicious tasks[4]. This communications may occur directly via a hard connection or indirectly via shared files or databases. The applications described in this paper use direct communications. Malicious activity typically is the sharing of private data, tracking user location, or interception of other applications' intents. This paper describes applications that provide the first two malicious activities. This paper also provides a brief overview of the related work of scanners which attempt to detect these malicious applications.

A. Class Work Requirements

The course work was to create at least two colluding Android applications that exhibit malicious activity when working together unbeknownst to the operator. These applications should have at least two components. Each application shall have allocated permissions suitable for the application that masks the malicious intent. Android restricts access to system API via permissions the user must explicitly grant[1]. The allocated permission for the malicious intent shall be different in the applications and shall be requested of the user who may or not grant such permission. Combined, these separate permissions provide a virtual permission set that facilitates the the malicious activity.

II. Applications

An application in a colluding system must be presented to look like a regular application to the user in order mask the true malicious intent of the developer; sending private data to the cloud. Each application has its own specific set of permissions the user must approve, so the application must give the appearance the permissions are actually needed so the application can perform its visible tasks. None have the same permissions. The applications created were named **NASA Client**, **BBStat**, and **Sundial**. Their definitions follow.

A. NASA Client

The NASA Client presents NASA pictures to the user. The application provides several ways to view the images. The first by default operation which shows *The Picture of the Day (POD)*. The second by the user selecting a particular date of interest for the POD. The third way is to select a date of interest which presents a set of Mars Rovers pictures. These activities mask the true intent of the developer; access to the INTERNET. It is the second action that initiates the upload of private data. Similarly, the *innocent* action of changing the volume triggers the upload, as well. Lastly, the same occurs when the system date changes. The system captures both the volume change and date change intents via a single *broadcast receiver*. The receiver is considered a component of the application[1]. These actions from user inputs or the receiver creates a connection to the BBStat application. Once a connection is established, NASA Client may send messages to BBStat. NASA Client receives messages via a service that provides an access point to BBStat and Sundial. See Figure 1 below.

A service is a separate component that runs in the application's background. The application's manifest defines this service as **exported**. In the Android framework, this allows other applications to communicate with this service via an **explicit intent**. (Android framework also provides for implicit intents, which where not used in this project, that are delivered to recipients at the discretion of the Android system.) The service is usually in a quiescent mode waiting for message bundles. A message bundle contains an ID and associated malicious data.

In this case, the ID defines the source application, BBStat or Sundial. After receipt of a message bundle, the service unpacks the bundle and sends the malicious data to the cloud.

A message passing system can become an attack surface if used incorrectly[1]. Thus, the methodology chosen can itself be susceptible to other rogue developers. In my design the message sender is protected, because it specifies the recipient through an explicit intent. However, since the recipient does not restrict who sends it messages, an attacker can also inject malicious messages into it[1].

1) Cloud Sink: The cloud sink is the destination of the malicious data. For this project, I used a PHP web service, **Jot**, hosted at jot.leseonline.net, developed for a previous class that provides a note storage service. NASA Client communicates with this web service storing the malicious data. For testing purposes, I developed a PHP web site that presents the uploaded data. See http://cs590.leseonline.net. Figure 2, below, shows an example of the web site presentation. Note the key contains the type of data and the uploaded timestamp.

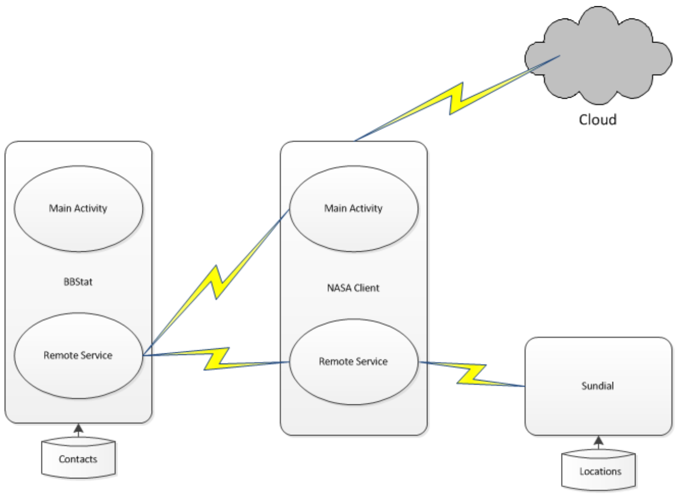
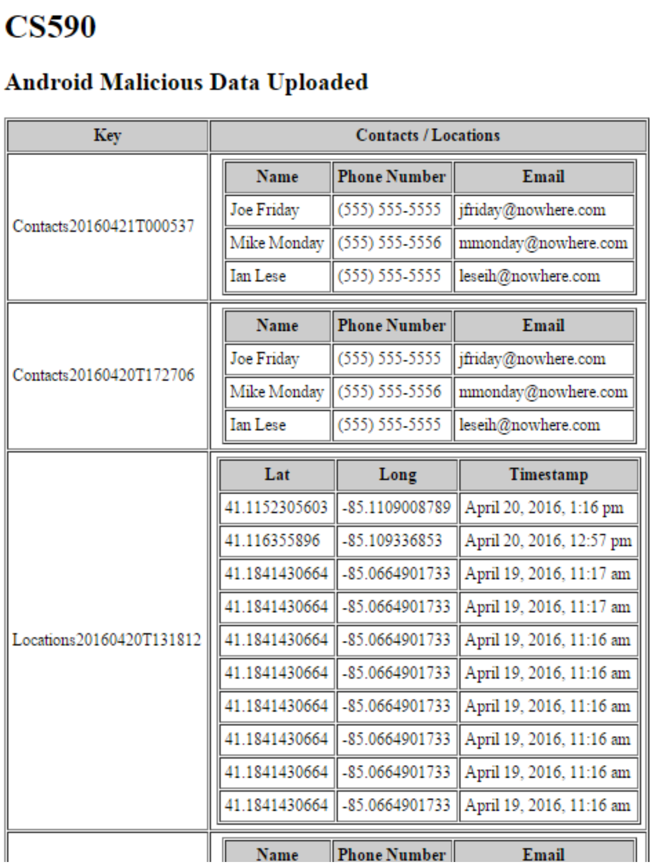


Fig. 1. A block diagram of the system.

B. BBStat

BBStat is a fake application that is in its UI infancy. This application masks its true intent by providing the user a means to enter basketball statistics and then share the results with others selected from the contact list. Thus, this application requests access to the contact list via the READ_CONTACTS permission. This application is rather benign, waiting for commands from an outside source, NASA Client in this system.

This application also contains a service. This service is actually started by the NASA Client main activity via an explicit intent creating an ICC portal for messages to move through. The BBStat service in turn starts the NASA client service via an explicit intent. When the BBStat service receives a given message ID (a number shared by both applications), it gathers the contacts list, creates a message bundle using the list, and then sends the message to NASA Client which then forwards the list to the cloud.



CS590

Android Malicious Data Uploaded

Key	Contacts / Locations		
Contacts20160421T000537	**Name**	**Phone Number**	**Email**
	Joe Friday	(555) 555-5555	jfriday@nowhere.com
	Mike Monday	(555) 555-5556	mmonday@nowhere.com
	Ian Lese	(555) 555-5555	leseih@nowhere.com
Contacts20160420T172706	**Name**	**Phone Number**	**Email**
	Joe Friday	(555) 555-5555	jfriday@nowhere.com
	Mike Monday	(555) 555-5556	mmonday@nowhere.com
	Ian Lese	(555) 555-5555	leseih@nowhere.com
Locations20160420T131812	**Lat**	**Long**	**Timestamp**
	41.1152305603	-85.1109008789	April 20, 2016, 1:16 pm
	41.116355896	-85.109336853	April 20, 2016, 12:57 pm
	41.1841430664	-85.0664901733	April 19, 2016, 11:17 am
	41.1841430664	-85.0664901733	April 19, 2016, 11:17 am
	41.1841430664	-85.0664901733	April 19, 2016, 11:16 am
	41.1841430664	-85.0664901733	April 19, 2016, 11:16 am
	41.1841430664	-85.0664901733	April 19, 2016, 11:16 am
	41.1841430664	-85.0664901733	April 19, 2016, 11:16 am
	41.1841430664	-85.0664901733	April 19, 2016, 11:16 am
	41.1841430664	-85.0664901733	April 19, 2016, 11:16 am
	Name	**Phone Number**	**Email**

Fig. 2. A snapshot of the uploaded data.

C. Sundial

Sundial is a simple application I developed many years ago, when the Android system was still young. Since a sundial's hour layout depends on latitude, this application must request the user to grant access to the device's location provider. This is a well-suited application to track a user's movement and was modified to do just that. This application contains a single activity that opens an ICC channel with the NASA Client service. Basically, Sundial reads the device's location every ten seconds. For simplicity, it stores the ten most recent locations in the SQLite database as latitude, longitude, and timestamp columns. If the most recent read location is the same as the next most recent, it is not stored. The application sends the locations in a message bundle to NASA Client when the user rotates the device. This application provides an indirect means between user action and ICC activity.

Sundial sets a *rotation flag* when the device is rotated. The application has a thread on a ten second cycle. The application reads a location each cycle and processes the location. During the cycle, the system checks the *rotation flag*. If set, the thread's worker sends the locations to NASA Client

III. Challenges

Creating Android application within the Android Studio environment was the most challenging part of the project. The

IDE was unfamiliar to me and posed a steep learning curve. The actual implementation of the colluding pieces was quite simple once I found examples that were similar to the threat model described by Elish, et.al.[4]

IV. Related Work

The work related to this project are papers which describe collusion techniques and scanners used to detect malicious colluding Android applications. An overview of two collusion techniques is given next followed by a short discussion of available scanners used to detect the colluding applications.

A. Collusion Techniques

Two collusion techniques are **Intent-Based Attack Surfaces** and **Overt and Covert Channels**.

1) Intent-Based Attack Surfaces: Intent-based attack surfaces are catorgorized as *Unauthorized Intent Receipt* and *Intent Spoofing*[1]. Unauthorized intent receipt happens when a malicious application intercepts an intent not intended for it. Recall, there is no guarantee the Android system will deliver an implicit intent to the intended party. Intent spoofing is when an explicit intent is sent to an exported component that is not expecting intents from the sending application[1].

There are several techniques described by Chin, et. al.[1] by which unauthorized intent receipt occurs. A few of the several are described here.

Broadcast Theft Broadcast Theft is a passive act of eavesdropping or an active act of denial of service. Eavesdropping poses a risk to the sender, because the application receives an intent that may contain private data (a bad programming practice by the sender developer to be sure).
Denial of service is a more complex process regarding *order broadcast*. An ordered broadcast is one where the intents are delivered in priority order. Any receiver can stop the broadcast at any time. The malicious application registers itself as a high priority receiver, thus receiving the broadcast before lower priority receivers. The application would then stop the broadcast, in effect denying service to other intended recipients.

Activity Hijacking In its simplest form, this malicious activity registers to receive the intent of another activity. When the intent is received, the malicious activity starts up and processes the intent. The malicious activity then responds to the sender. Since this involves an activity, the user may be asked to decide what application should receive the intent. There are more sophisticated examples not discussed here.

Service Hijacking This scenario is very similar to the Activity Highjacking, but involves services. The user is unaware of this, since there is no user-interface involved. Once a connection is made between the sender and the malicious service, the service can send messages or responses back to the sender or do whatever with the data.

2) Overt and Covert Channels: More apropos to the applications created for this project, stealthy communications channels may be created to share data. Marforio, et. al. describes the property of these channels to be overt or covert[2].

Overt Channels An overt channel is a stealthy connection between two colluding applications. The applications created for this project use overt channels through the explicit intents and the broadcast receiver. Another means, not used in this project, is the use of the system log. In this case, the source applications writes data to be shared to the system log and the sink application reads and processes the data. The last overt style channel utilizes the well-known UNIX socket communications between the two colluding applications.

Covert Channels A covert channel uses similar means as overt, but because the channel is operated synchronously, both applications must be active at the same time. This is not always the case for an overt channel. I have to admit, the covert channels have me somewhat befuddled, except for the broadcast intent. The broadcast intent is nearly identical to the overt, but the broadcast data is encoded for use by the receiver. Most other covert channels use low level features of the operating system in ways not typical of other Android applications.

B. Scanners

This section give a brief overview of scanners which try to identify potential malicious applications. Two tools that are several years old are TaintDroid and XManDroid. A more recent profiling tool created by Elish, et. al.[3] reduces the number of false positives reported by previous tools.

TaintDroid TaintDroid uses a modified Android OS. This tool tracks private data from the source to the receiver. TaintDroid notifies the user when it believes the application is leaking data to other than the intended party. This tool does not work well with covert channels.

XManDroid XManDroid also is a modification of the Android OS. This tool is well-suited for detecting most overt channel collusion. XManDroid is also useful to detect some covert channels (low-level OS) operations as well. This tool also reports false-positives, legitimate channels that share data. Marforio, et. al. describe XManDroid as reactionary, since hooks are added to the OS calls when new channels are discovered[2].

Elish, et. al. (unnamed) The tool developed by Elish, et. al. performs a static analysis of applications. The tool extracts definition-use data dependence properties related to sensitive operations, typically API calls. The extracted data is used to create a Data Dependence Graph that tracks the data from some trigger to its sensitive destination. Nodes of the graph are weighted and accumulated resulting in the decision of a valid or an invalid data dependency path. [Sundial contains an invalid path where a trigger (the device rotation) is completed disconnected (in my view) from the call that sends the data to NASA Client.] This tool reports very few false-positives compared to most other similar tools.

V. Future Work

This project proves the Android framework provides a rather easy way to create colluding applications. What it did not prove is whether these applications can thwart the malware detection scanners. If there is any future work to be done, it will entail scanning these applications. Since I do not believe I am clever enough to create un-detectable applications, these current applications will be hardened or new ones will be created with the specific intent to thwart the scanners, thus providing the industry an opportunity to enhance the scanning techniques. It would be of interest to know if Elish, et. al.'s tool would find the Sundial to be a malicious application, since its trigger and output channel are disconnected.

It would be worthwhile to develop applications that include many of the channels described by Marforio, et. al.[2] and attack surfaces described by Chin, et. al[1].

VI. Conclusion

The Android framework restricts access to the system API through permissions granted by the user. The framework allows for inter-component communication to provide rich application development. Unfortunately, this allows a means for a developer to create several colluding applications producing a virtual permission set that enables malicious activity. The applications developed for this project prove malicious application development is quite simple and can be used to share the private data of unsuspecting users.

Acknowledgment

I would like to thank Dr. Karim Elish for his guidance during the exercise of this project.

References

[1] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner, Analyzing Inter-Application Communication in Android New York, NY: MobiSys '11 Proceedings of the 9th international conference on Mobile systems, applications, and services, 2011

[2] Claudio Marforio, Hubert Ritzdorf, Aurelien Francillon, and Srdjan Capkun, Anaylysis of the Communication between Colluding Applications on Modern Smartphones New York, NY: ACSAC '12 Proceedings of the 28th Annual Computer Security Applications Conference, 2012

[3] Karim O. Elish, Xiaokui Shu, Dafeng Yao, Barbara G. Ryder, and Xuxian Jiang, Profiling user-trigger dependencies for Android malware detection Oxford, UK: Computers and Security archive Volume 49 Issue C, Elsevier Ltd., March 2015

[4] Karim O. Elish, Dafeng Yao, and Barbara G. Ryder, On the Need of Precise Inter-App ICC Classification for Detecting Android Malware Collusions In Proceedings of IEEE Mobile Security Technologies (MoST), in conjunction with the IEEE Symposium on Security and Privacy, 2015.