Android App Collusion

Ervin Oro
ervin.oro@aalto.fi

Tutor: Jorden Whitefield

set proper margins and document format

Abstract

abstract

KEYWORDS: *list, of, key, words*

1 Introduction

Android is an operating system (OS) that is primarily designed for mobile devices, e.g., smartphones and tablets. With more than two billion active devices [1], it is estimated to be the most widely used OS, surpassing Microsoft Windows [2, 3]. Android is designed to be an open platform: developed and maintained by Google LLC, but largely released as the Android Open Source Project (AOSP) for everyone to study and evaluate [4]. Android OS includes support for apps, which are easily installable application packages that can extend the functionality of devices. Apps can be developed and distributed by anyone with very low barrier of entry.

While this popularity of Android is not reflected by the proportion of malware attacks, most of which still target Windows, both the number and complexity of attacks against Android are increasing [5]. This is especially

troublesome as many people increasingly rely on their smartphones – often to store their personal data, online account credentials, money, and more. McAfee estimates that revenues for mobile malware authors could be in the billion-dollar range by 2020 [6].

Given the increasing potential damage from Android malware, defending against it is an active area of research. Android uses a multi-layer security approach, combining machine learning, platform security and secure hardware [1]. Machine learning methods are utilised by Google Play Store to prevent uploading potentially harmful applications, and by Google Play Protect[1] to scan apps locally on users' devices. Android's platform security has been enhanced over the years with the addition of multiple security features, for example, SELinux protections[2], exploit mitigations[3], privilege reductions[4], and encryption. Recent versions of Android leverage hardware security features, including keystore and remote key attestation[5], and receive regular software updates. These security mechanisms have been partially successful, as exploit pricing and difficulty are growing by some estimates [1].

citation vs footnote

However, malicious actors are continuously developing exploits to bypass existing protections, and a number of threats, e.g., app collusion, cannot yet be reliably detected nor defended against. App collusion is a secret collaboration between apps with malicious intent (Section 2). This can be facilitated by any of the numerous ways for apps to communicate with each other that the Android system provides (Section 3). Methods for apps to collude also exist on the iOS platform [7]. Given a malicious app that would be detected and blocked with state of the art security systems, its functionality can be split into several apps, so that each of them would be categorised as benign when analysed separately [8].

Android app collusion is not a new concept [9], and multiple attempts have been made to develop a suitable detection system.

brief overview of existing approaches based on Section 5

Despite this, there are currently no robust and usable ways to detect app collusions. Most proposed solutions have a large number of false positives due to their inability to differentiate collusion from legitimate

[1] https://www.android.com/play-protect/

[2] https://source.android.com/security/selinux/

[3] https://lwn.net/Articles/695991/

[4] https://android-developers.googleblog.com/2017/07/seccomp-filter-in-android-o.html

[5] https://android-developers.googleblog.com/2017/09/keystore-key-attestation.html

collaboration. Furthermore, since the number of possible combinations is exponential in the number of apps, that is, N^N, most proposed solutions apply very aggressive filtering, causing only some malicious combinations to be included into analysis, and others to be reported as false negatives. ~~Finally,~~ approaches attempting to overcome both of these issues have been computationally infeasible thus far. Therefore, app collusion remains an open research challenge.

explain why N^N

This report aims to provide an overview of app collusion on the Android platform as follows. Section 2 discusses the nature of app collusions in general, Section 3 provides specific overview of methods that can be used for colluding on Android, Section 4 describes known examples of colluding apps, and Section 5 gives a more in-depth systematic overview of approaches that have been taken to collusion detection and their limitations.

2 Description and definition of app collusion

The Oxford English Dictionary defines collusion as "Secret agreement or understanding for purposes of trickery or fraud; underhand scheming or working with another; deceit, fraud, trickery" [10]. Asăvoae et al. [11] define collusion for the case of Android apps as the situation where several apps are working together in performing a threat. According to these definitions, app collusion must have the following three properties:

1. Colluding apps must be working together secretly. Conversely, apps working together in collaboration is common and encouraged practice when such collaboration is well documented [!] .

Citation needed

2. All colluding apps must be in agreement. A distinctly different but related concept is the "confused deputy" attack, where one app mistakenly exposes itself to other installed apps [!] .

Hardy, N.: The confused deputy:(or why capabilities might have been invented. ACM-SIGOPS Operating Systems Review 22(4), 36–38 (1988)

3. Colluding apps must have malicious intent. The intent of Android app collusion would then be to violate one of Android's security goals, which are defined in [12] as:

   (a) to protect app data, user data, and system resources (including the network),

I also wonder if it is worth men-

(b) to provide app isolation from the system, other apps, and the user.

It is important to note that the goal 3(b) is not to enforce isolation, but ~~merely~~ to provide isolation for those who require it. As such, apps working together do not automatically violate goal 3(b), but it would be collusion if apps worked together to break isolation with some other non-content app, the system, or the user.

many things affect, including non technical

difficult to distinguish

alternative definition [13]

3 Methods for colluding

By default, all Android apps run in separate sandboxes [14], which are based on user separation by the Linux kernel, and enhanced by SELinux and `seccomp` [15]. By default, all communication between sandboxes is blocked [14], but apps can open certain communication channels or prevent being separated into different sandboxes altogether. Some channels, so-called overt channels, are designed to be used by apps to communicate, while others, so-called covert channels, utilise functionalities intended for other purposes. All channels discussed below require the participation of both parties, but some researchers have looked into ways to cross sandbox borders unilaterally, therefore breaking the goal 3(b) in section 2 [!] .

Citation needed

3.1 Overt channels

Android Open Source Project describes channels designed for inter-app communication, such as sandbox sharing and binder, in [14] and [15].

Apps published by the same entity may share a sandbox. In this case, there are no restrictions for their communication. These apps can use any of the traditional UNIX-type mechanisms, including filesystem, local sockets, or signals.

When apps are running in different sandboxes, the Linux kernel prevents these sandboxed apps from accessing each other's processes or files. In older Android versions, only Linux discretionary access control was used, allowing apps to make their files world-accessible, but newer versions of Android forbid this using SELinux mandatory access control rules. Apps can still use any file-based communication methods when they have

permission to access the external storage, but this way users would have some visibility into the fact that such communication channels may be used by the app.

However, Android also provides a method for apps to communicate without any user-granted permission or visibility. This is enabled by a remote procedure call mechanism called binder. Any app can send messages to the binder arbitrarily, but other apps must explicitly start listening and accept incoming communications. The Android platform provides three main ways to do this:

1. Services [!] . Apps may start services, which can provide interfaces that are directly accessible using binder.

2. Intent filters [!] . Intents are simple message objects that represents an "intention" to do something. Apps may ask some part of them to be executed when an intent with certain properties matching their filter is initiated.

3. ContentProviders [!] . Apps can define ContentProviders to expose some of their data.

maybe add something about android platform architecture

The binder provides an easy way for apps to communicate with each other, promoting openness and allowing separation of concerns. Examples include apps using an intent to ask the camera app to take a photo instead of asking camera control permission, and communication apps allowing other apps to share data through itself. Binder has a well-defined interface that could be monitored.

3.2 Covert channels

In addition to the overt channels, a large amount of covert channels have been discovered. Marforio et al. [16] propose a classification of communication channels based on whether Application level APIs, OS native calls, or Hardware functionalities are utilised. Al-Haiqi et al. [17] describe categorising covert channels as either timing or storage channels. However, neither of these approaches provides clear boundaries in all cases nor covers all possible covert channels, which by their nature form an unbounded set. This section provides some examples of covert channels in Android.

revisit after section 5: if it discusses more about binder; it needs to be introduces more - a short introduction to Android platform architecture with figure. See `https://developer.android.com/guide/platform`

read more

read more

See `https://developer.android.com/guide/components/intents-filters`. Include Figure 1. Explicit/implicit intents. Look at warning boxes, some interesting advice.

read more

Schlegel et al. [9] show that any application can change the vibrate setting and use intent filters to be notified every time the setting is changed without requiring specific permissions, therefore demonstrating that apps can create an information channel using the vibrate setting. Similarly, the volume setting can be used. While apps cannot subscribe to be notified when volume is changed and have to manually check this setting, it has the benefit of having 8 different states, as opposed to the vibrate setting, which is a boolean value. These channels are invisible to users, as long as data transmission does not coincide with audio playback or receiving notifications.

Marforio et al. [16] describe how data could be exchanged between colluding apps by modifying and monitoring the number of threads, processor usage, and free space on filesystem. While some of the APIs they used have been deprecated [18], partially similar approaches are possible on modern Android versions as well. For example, free disk space can be queried through the StatFs#getAvailableBlocksLong() API on latest Android versions [19].

signpost novel idea

The system load can also be measured indirectly to transmit information [16]. In this scenario, the transmitting app modulates data payload by varying the load on the system. The Receiving app then repeatedly runs a CPU-intensive computation and measures the time it takes to complete, in order to infer whether or not the transmitting app is loading the system or not. This approach was shown to work even when receiver is just some JavaScript in the browser and not an installed Android app.

Another approach is presented by Al-Haiqi et al. [17], where one app utilises the vibration motor to transmit data, and another app uses the accelerometer readings to receive that data. This is further developed by Qi et al. [20], who propose covert channels based on user behaviour. Instead of using the vibration motor, a transmitting app could prompt the user to move their phone in certain ways, for example, posing as a rally game where the user needs to turn their phone at specific times based on a track generated by the malicious app.

Add a subsection conclusion paragraph

4 Examples of Android app collusion

A basic example of app collusion (Figure 1) is as follows:

1. APP$_A$ obtains some private information.
2. APP$_A$ transmits the information to APP$_B$ using some overt or covert channel.
3. APP$_B$ exfiltrates the information over the internet.

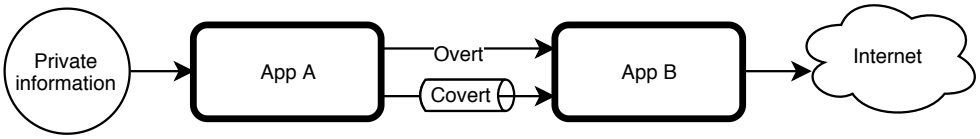

Figure 1. Structure of a basic app collusion example.

An early example of this kind of collusion was described by Schlegel et al. [9]. In their case, APP$_A$ was the Soundcomber app that obtained private information using the microphone. To avoid detection, the Soundcomber app did not have permission to access the internet, so they proposed to use a second app to exfiltrate the information. Similar hypothetical example is also described by Asăvoae et al. [11], where APP$_A$ would be a contacts app with READ_CONTACTS permission, and APP$_B$ would be a weather app with INTERNET permission.

MoPlus SDK

5 Existing methods for detecting collusions

Bibliography

[1] Android Open Source Project *et al.*, "Android security 2017 year in review," Mar. 2018. [Online]. Available: https://source.android.com/security/reports/Google_Android_Security_2017_Report_Final.pdf

[2] Awio Web Services LLC, "Browser & platform market share," Dec. 2018. [Online]. Available: https://www.w3counter.com/globalstats.php?year=2018&month=12

[3] StatCounter, "Operating system market share worldwide," Dec. 2018. [Online]. Available: http://gs.statcounter.com/os-market-share

[4] Android Open Source Project, "Legal notice," 2019. [Online]. Available: https://developer.android.com/legal

[5] AV-TEST GmbH, "Security report 2017/18," Jul. 2018. [Online]. Available: https://www.av-test.org/en/news/the-av-test-security-report-20172018-the-latest-analysis-of-the-it-threat-scenario/

[6] McAfee, "Mobile threat report," Apr. 2018.

[7] L. Deshotels *et al.*, "SandScout," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16*. ACM Press, 2016. [Online]. Available: https://dl.acm.org/citation.cfm?id=2978336

[8] H. Chen *et al.*, "Malware collusion attack against machine learning based methods: issues and countermeasures," in *Cloud Computing and Security*, X. Sun, Z. Pan, and E. Bertino, Eds. Cham: Springer International Publishing, 2018, pp. 465–477. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-00018-9_41

[9] R. Schlegel *et al.*, "Soundcomber: A stealthy and context-aware sound trojan for smartphones," in *Proceedings of the Network and Distributed System Security Symposium, NDSS'2011*, Jan. 2011. [Online]. Available: https://www.researchgate.net/publication/221655538_Soundcomber_A_Stealthy_and_Context-Aware_Sound_Trojan_for_Smartphones

[10] "collusion, n." in *OED Online*. Oxford University Press, Dec. 2018. [Online]. Available: http://www.oed.com/view/Entry/36460

[11] I. M. Asăvoae *et al.*, "Detecting malicious collusion between mobile software applications: the Android™ case," in *Data Analytics and Decision Support for Cybersecurity*. Springer International Publishing, Aug. 2017, pp. 55–97.

[12] Android Open Source Project, "Security," 2019. [Online]. Available: https://source.android.com/security

[13] M. Xu *et al.*, "AppHolmes," in *Proceedings of the 26th International Conference on World Wide Web - WWW '17*. ACM Press, 2017. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7349085/

[14] Android Open Source Project, "Application security," 2019. [Online]. Available: https://source.android.com/security/overview/app-security

[15] ——, "Application sandbox," 2019. [Online]. Available: https://source.android.com/security/app-sandbox

[16] C. Marforio *et al.*, "Analysis of the communication between colluding applications on modern smartphones," in *Proceedings of the 28th Annual Computer Security Applications Conference on - ACSAC '12*. ACM Press, 2012. [Online]. Available: https://dl.acm.org/citation.cfm?id=2420958

[17] A. Al-Haiqi, M. Ismail, and R. Nordin, "A new sensors-based covert channel on android," *The Scientific World Journal*, vol. 2014, pp. 1–14, 2014. [Online]. Available: https://www.hindawi.com/journals/tswj/2014/969628/abs/

[18] nn...@google.com, "Android O prevents access to /proc/stat," Mar. 2017. [Online]. Available: https://issuetracker.google.com/issues/37140047

[19] Android Open Source Project, "API reference," 2019. [Online]. Available: https://developer.android.com/reference

[20] W. Qi *et al.*, "Construction and mitigation of user-behavior-based covert channels on smartphones," *IEEE Transactions on Mobile Computing*, vol. 17, no. 1, pp. 44–57, jan 2018. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7908988

[21] F. I. Abro *et al.*, "Android application collusion demystified," in *Future Network Systems and Security*, R. Doss, S. Piramuthu, and W. Zhou, Eds. Cham: Springer International Publishing, 2017, pp. 176–187. [Online]. Available: http://openaccess.city.ac.uk/18503/

[22] I. M. Asavoae *et al.*, "Towards automated Android app collusion detection," *arXiv preprint arXiv:1603.02308*, 2016. [Online]. Available: https://arxiv.org/abs/1603.02308

[23] I. M. Asăvoae, H. N. Nguyen, and M. Roggenbach, "Software model checking for mobile security – collusion detection in $\mathbb{K}$," in *Model Checking Software*, M. d. M. Gallardo and P. Merino, Eds. Cham: Springer International Publishing, 2018, pp. 3–25. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-94111-0_1

[24] J. Blasco *et al.*, "Wild Android collusions," in *VB2016*, Oct. 2016. [Online]. Available: https://www.virusbulletin.com/conference/vb2016/abstracts/wild-android-collusions

[25] H. Chen *et al.*, "Malware collusion attack against SVM: issues and countermeasures," *Applied Sciences*, vol. 8, no. 10, 2018. [Online]. Available: http://www.mdpi.com/2076-3417/8/10/1718

[26] D. Davidson *et al.*, "Enhancing Android security through app splitting," in *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*. Springer International Publishing, 2018, pp. 24–44.

[27] K. Elish *et al.*, "Identifying mobile inter-app communication risks," *IEEE Transactions on Mobile Computing*, 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8587187

[28] Google, Inc., "The Google Android Security Team's classifications for potentially harmful applications," Feb. 2017. [Online]. Available: https://source.android.com/security/reports/Google_Android_Security_PHA_classifications.pdf

[29] McAfee, “Safeguarding against colluding mobile apps,” May 2016. [Online]. Available: https://www.mcafee.com/enterprise/en-us/assets/solution-briefs/sb-quarterly-threats-may-2016-1.pdf

[30] I. Muttik, “Partners in crime: investigating mobile app collusion,” in *McAfee Labs threats report*. McAfee, Jun. 2016, pp. 8–15. [Online]. Available: https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-may-2016.pdf

[31] Android Open Source Project, “The Android source code,” 2019. [Online]. Available: https://source.android.com/setup

[32] L. Qiu, Y. Wang, and J. Rubin, “Analyzing the analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe,” in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis - ISSTA 2018*. ACM Press, 2018.

[33] F. Wei *et al.*, “Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps,” *ACM Transactions on Privacy and Security*, vol. 21, no. 3, pp. 1–32, apr 2018.

[34] J. Zhan *et al.*, “Splitting third-party libraries’ privileges from Android apps,” in *Information Security and Privacy*. Springer International Publishing, May 2017, pp. 80–94.

format bibliography

remove nocite

Todo list

remove list of todos