Transplantation Attack: Analysis and Prediction

Zhongwen Zhang[1,2,3], Ji Xiang[1,2](✉), Lei Wang[1,2], and Lingguang Lei[1,2]

[1] Data Assurance and Communication Security Research Center, Beijing, China
[2] Institute of Information Engineering, CAS, Beijing, China
[3] University of Chinese Academy of Sciences, Beijing, China
{zwzhang,jixiang,lwang,lglei}@lois.cn

Abstract. Correspondingly, Android also becomes a common attack target. Till now, many attacks have been detected out, such as confused deputy attack, collusion attack, and root exploits attack. In this paper, we present a novel attack, denoted as transplantation attack. Transplantation attack, when being applied to spy on user, can make the malicious behavior more stealthy. The attack can evade permission check, evade device administration, and even evade API auditing. The premise of carrying out Transplantation attack is that malware is able to access resources or gain access capability. By fulfilling the premise, we do a case study about Camera device. The result indicates that Transplantation attack indeed exists. Based on these observations, we predict the kind of system resources that may suffer transplantation attack. Defence discussion are also presented.

Keywords: Android · Transplantation Attack · Prediction

1 Introduction

Nowadays, Android becomes the most wide spread mobile platform. In the meanwhile, it also becomes a common attack target. Many kinds of attacks towards Android system have been detected out, e.g., confused deputy attack [5,8,9,12], collusion attack [4], and root exploits attack [17,24]. In this paper, we will present a novel kind of attack, and we name it as transplantation attack.

To explain what is transplantation attack, we should mention the resource accessing procedure first. In Android system, most system resources (e.g., GPS, camera) are accessed by system services (e.g., LocationManager Service, Camera Service). Applications (apps), if want to access these resources, should send request to system services via IPC (Inter-Process Communication). Then, system services will call several system libraries (*.so* libraries) to talk with resource driver and collect data and return data back to apps. When resources are accessed, two

This work is supported by National Natural Science Foundation of China grant 70890084/G021102 and 61003274, Strategy Pilot Project of Chinese Academy of Sciences sub-project XDA06010702, and National High Technology Research and Development Program of China (863 Program, No. 2013AA01A214 and 2012AA013104).

J. Tian et al. (Eds.): SecureComm 2014, Part II, LNICST 153, pp. 367–374, 2015.
DOI: 10.1007/978-3-319-23802-9_28

processes are involved, an application's (client) process and a system service's (server) process. In this case, system *.so* libraries run in system services' address space.

However, what if malware transplant system *.so* libraries from system services' address space to their own address space, and use these libraries to talk with driver to collect data by their own? In this case, system *.so* libraries will run in malware's address space, which will lead to several security issues; and it should be considered as an attack. We call this attack as **Transplantation Attack**.

Transplantation attack enables malware accessing resources without involving IPC with system services. In transplantation attack, when resources are accessed, only one process (app's process) is involved, and there is nothing to do with system services' process. Therefore, a lot of security enforcements implemented in system services can be evaded.

By starting a transplantation attack, malware can evade permission check. In Android system, before system services respond to resource accessing request of an app, they will check the app's permissions first. If the app does not have the required permission, system services will not provide service. In transplantation attack, malware do not depend on system services to get data, instead, they collect data by their own. Therefore, the permission check process initiated by system services can be evaded in transplantation attack.

By starting a transplantation attack, malware can evade device administration. Android framework provides several special Android APIs, which are called Device Administration APIs. They could be used by device admin apps to configure a phone. Device administration is usually enforced in enterprises, in which phones are used to do business. Usually, enterprise administrators install device admin apps on these phones to protect commercial benefit. Transplantation attack will make these admin apps fail to be effective. That is because, the device administration is also implemented in system services which are not called in transplantation attack.

By starting a transplantation attack, malware can evade API auditing. Android API auditing is important to both enterprise environments and individual users. In enterprise environments, deploying API auditing on employee phones helps to decrease security risk. For individual users, API auditing helps to detect spyware. API auditing can be done when permissions are checked. Since there is no permission check step in transplantation attack, malware can evade API auditing as well.

As a result, transplantation attack, when being applied to spy on user, can make malicious behavior much more stealthy and much more difficult to detect. We have searched the CVE (Common Vulnerabilities and Exposures) list [1]. Among the 448 CVE entries that match the keyword *Android*, we find there was no such kind of attack happened before.

In this paper, we will give an overview about transplantation attack. First we will describe the premise to start transplantation attack. Then, by fulfilling the premise, we carry out a transplantation attack towards Camera device as a case study. The case study verifies that transplantation attack indeed exists, and cannot be detected out by Antivirus. Base on these observations, we predict that other resources may also suffer transplantation attack. Moreover, we discuss potential solutions against this attack.

2 The Premise of Transplantation Attack

Transplantation attack is to transplant system libraries from system services' address space to malware's address space, and then malware can collect data in its own address space.

A big premise for the transplantation attack to be carried out is that the system libraries are designed to be called by everyone. However, that malware can call system libraries does not mean the malware can successfully access resources (e.g., hardware drivers, database files). Another premise of transplantation attack is that the malware itself should be able to access resources or can gain access capability.

System resources can be divided into two types: hardware resources and software resources. The way to achieve the premise of the attack towards the two kind resources is different.

Hardware Resources. To access hardware resources, e.g., GPS, Camera, malware should be able to access hardware drivers. Hardware drivers subject to Linux file system access control. To access a hardware driver, a user (app) should be the owner or be a member of the hardware's group. In Android system, an app could be a member of a hardware's group, aka., the app could be assigned with the hardware's group id (GID).

Apps could become a group member of some hardware through obtaining a certain permission. That is because, Android has bound some permissions with some groups, which are recorded in a metadata file (*platform.xml*). If a permission has been bound to a group, then once this permission is granted to an app, the app will be automatically set as a member of the group. By applying a permission, an app can gain the corresponding GID. After gaining the GID, an app could access the corresponding hardware driver. Once an app could directly access a hardware driver, it can start transplantation attack.

Software Resources. Most software resources, e.g., SMS, Contact, social network data, exist in the form of database files or in shared memories, which are files, too. These files are owned by system apps (e.g., SMS app, Contact app) or third party apps. To access these files, malware should become a shared user with the owner of these files, aka, the malware should share UID with the owner.

Sharing UID with a system app cannot be achieved except exploiting system vulnerabilities. Nowadays, two vulnerabilities towards signature verification have been detected out in Android system [18,21]. Exploiting them, malware is able to share UID with a system app, or a third party app. To share UID with a third party app, collusion attack also is an optional way. Once an app becomes a shared user of the file owner, it can access the file. As long as malware can access resource files, it can start transplantation attack.

Others. In one case that malware neither need to become a group member nor need to become a shared user. That is, a file, either a device file or a regular file, is publicly accessed. For example, the file's access rule is set as 666 (rw-rw-rw-) or 777 (rwxrwxrwx).

An advantage of attacks towards software resources is that malware does not need to apply any permission for any reason. On the contrary, attacks towards hardware resources should apply permissions to get GIDs. Therefore, transplantation attacks towards software resources are much more stealthy and much more hard to detect than attacks towards hardware resources.

3 Case Study

To verify the feasibility of transplantation attack, we have done a case study towards Camera device. We use a malicious app to carry out a transplantation attack.

As described before, the attack just transplants necessary *.so* libraries from Camera Service's address space to the malicious app's address space, and calls picture taking function provided by these *.so* libraries to take a picture. To make the malicious app be able to access camera driver, we should put the app into *camera* group. It can be achieved by applying CAMERA permission, because Android has bound CAMERA permission with *camera* group.

After going through a lot of failures, we successfully conduct a way of picture taking inside the malicious app's address space. Also, the malicious app is successfully executed on Nexus S with Android version 4.0.4 and Sony LT29i with Android version 4.1.2.

We also tested whether the malicious app can evade detection of Antivirus and can evade enterprise device administration. The test result shows both of them can be evaded.

4 Prediction of Transplantation Attack

The case study about Camera device indicates that transplantation attack indeed exists. In this section, we will predict where transplantation attack may happen.

4.1 Attack Towards Hardware Resources

As described before, malware should be able to access a hardware before starting transplantation attack. This can happens in two situations. One situation is that the malware is able to gain the GID of a hardware. The publicly available GIDs are recorded in the *platform.xml* file. The other situation is that a hardware is publicly accessed. For example, a hardware's access rule is 666. Hardware covered by the two ways are vulnerable to suffer transplantation attack.

Take the *platform.xml* file on Galaxy Nexus of version 4.1.2 as an example, available GIDs are *net_bt_admin, net_bt, inet, camera, log, sdcard_r, sdcard_rw,* *media_rw, mtp, net_admin, cache, input, diag, net_bw_stats, net_bw_acct*; and 15 of them in total. Hardwares involved in these GIDs are vulnerable to suffer transplantation attack, such as Bluetooth (GID: *net_bt_admin, net_bt*), Internet (GID: *inet*), Camera (GID: *camera*). The GPS' group id is not publicly available, so GPS does not suffer this attack.

Taking advantage of the transplantation attack, malware could enjoy Bluetooth stealthily. Nowadays, Bluetooth is commonly used in e-health area such as blood pressure monitor, glucometer, and wearable devices such as watches, glasses. Malware may steal this kind of high sensitive data without leaving any record.

Sometimes, vendors mistakenly configure a phone [23]. For example, on Samsung GT-I9300 phone, the GID *radio* is recorded in the *platform.xml* file as well. By gaining the GID *inet* and *radio*, malware may be able to use Internet without being detected. Attacks towards Internet will lead to users' financial lose.

Besides hardware whose GID can be applied may suffer the attack, mistakenly configured hardware may suffer the attack, too. According to [23], vendors set access rules of some hardware drivers on some phones as 666, which means all users can read and write these drivers. On these phones, malware can directly access those mistakenly configured hardware drivers without gaining their GID, aka, without applying any permission.

4.2 Attack Towards Software Resources

Software resources like private database files or shared memories are owned by apps. The GID of these files are not bound with permissions. Therefore, unless these files can be publicly accessed, malware should become a shared user to initiate attacks on them. It is designed that shared users can access each other's data, and, if desired, run in the same process. So, once malware becomes a shared user, it seems that it is not necessary to do transplantation attack.

However, regular access way may leave auditing record. For example, database files are regularly accessed via Content Provider, which can be used to do API auditing. Moreover, other shared users may do extra access control or do auditing inside their execution flows, too. These obstacles may become a motivation of carrying out a transplantation attack towards software resources.

In case of malware sharing UID with other apps contain private data, we predicate that transplantation attack may happen.

5 Defence Discussion

There may be several ways to defend the transplantation attack, but some of them may not work out. For example, forbidding the usage of system libraries may sound a good idea to defend the attack. However, as apps can ship their own copy of the required system library, this way may not work out. Here, we discuss two possible ways as follows.

5.1 Breaking the Binding Between Permission and Group ID

The transplantation attack should get the capability of accessing a hardware device. To gain this capability, the malicious app should be assigned with the hardware's group ID by applying corresponding permission. Noticing this, a defence is that we could break the binding between permission strings and group ids. Taking Camera device as an example, breaking the binding between camera permission and camera group id will not affect the normal apps to take pictures. That is because Camera device has a daemon process (*mediaserver* process, in which Camera Service runs) in charge of taking pictures. Apps just need to send request to the daemon process, and the process will handle the picture taking. One weakness of this defence is that when the hardware has zero daemon process (e.g., Sdcard) or more than one daemon processes, it is possible to result in denying of services.

5.2 Using SEAndroid Policy

SEAndroid enforces mandatory access control to every process (user) under a fine-grained access control policy. Every process belongs to a domain (type). Here, third-party apps are classified into the *untrusted_app* domain, which will be blocked when directly access protected files (device files, regular files).

Although SEAndroid can block the access to protected files, it has a rather limited enforcement range. SEAndroid [20] is merged into AOSP since version 4.3 and enforced since version 4.4. According to Google's survey [14], the phones shipped with version 4.3 and 4.4 each accounts for 8.5 % of the total at the beginning of May, 2014. That a phone shipped with 4.3 version of Android does not mean that the SEAndroid is enforced. So, nearly 90 % of the Android phones in the wild are however not protected by SEAndroid. Among the phones used by our labmates, 93 % of them without SEAndroid. It may take a long period of time before SEAndroid can be widely deployed in the wild. During this period of time, many users may suffer from the spy-on-user attack.

Besides the distribution range limitation, SEAndroid has weakness as well. Pau Oliva shows three weaknesses of SEAndroid and gives out four ways to bypass SEAndroid [22]. We did an experiment, in which we change SEAndroid from enforce mode to permissive mode via PC terminals. The same principle could be applied to apps. The experiment shows that SEAndroid can indeed be bypassed.

6 Related Work

Confused Deputy Attacks. Confused deputy attack means a malicious app without permission P exploits the unprotected interfaces of other apps with permission P to perform a privileged task for itself. To detect whether an app has unprotected interfaces, a number of detection tools have been proposed [2,6,11,13]. These static analysis tools are likely to be incomplete, as they cannot completely predict the actual confused deputy attack occurring at runtime.

To address this issue, some framework extension solutions [9,12] have been proposed.

Collusion Attacks. Different from the confused deputy attacks, the collusion attacks concern malicious apps that collude to combine their permissions. So, one malicious app does not need to apply all permissions, which can evade the detection of Kirin [10]. To address the collusion problems, [3,4,15] are proposed. These solutions can confront both the deputy attacks and the collusion attacks.

Root Exploits Attacks. According to [24], attacks exploiting root privilege play a significant role in compromising Android security. Among the root exploiting malware, the *DroidKungFu* [17] is a typical example. Attacks exploiting root privilege could break the boundary of Android sandbox and could access resources without applying permissions. The root exploits attacks could be blocked by SEAndroid [20]. By introducing SEAndroid, processes even running with root privilege cannot access the protected files and devices.

Security Enhancements. Some framework security extension solutions [7,16,19,25] enforce runtime permission control to restrict apps' permissions at runtime. These solutions aim at providing a fine-grained access control for IPC. The novel transplantation attack does not call Android APIs or does not involve IPC. Therefore, these solutions cannot block the transplantation attack.

7 Conclusion

In this paper, we give an overview about the transplantation attack, which can make malicious behavior much more stealthy when being applied to spy on user. We first describe the premise of the attack, then we do a case study on Camera device, which verifies the attack indeed exists. Based on the premise and case study result, we predict that there are other resources may suffer transplantation attack. At last, we discuss potential defences towards the attack.

References

1. CVE: Common vulnerabilities and exposures. http://cve.mitre.org/
2. Au, K.W.Y., Zhou, Y.F., Huang, Z., Lie, D.: Pscout: analyzing the android permission specification. In: ACM CCS (2012)
3. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.R.: XMandroid: a new android evolution to mitigate privilege escalation attacks. Technische Universität Darmstadt, Technical Report TR-2011-04
4. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.R., Shastry, B.: Towards taming privilege- escalation attacks on android. In: 19th NDSS 2012 (2012)
5. Chan, P.P., Hui, L.C., Yiu, S.: A privilege escalation vulnerability checking system for android applications. In: 2011 IEEE 13th International Conference on Communication Technology (ICCT), pp. 681–686. IEEE (2011)
6. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in android. In: 9th MobiSys 2011 (2011)

7. Conti, M., Nguyen, V.T.N., Crispo, B.: Crepe: context-related policy enforcement for android. In: Information Security (2011)
8. Davi, L., Dmitrienko, A., Sadeghi, A.R., Winandy, M.: Privilege escalation attacks on android. In: Burmester, M., Tsudik, G., Magliveras, S., Ilić, I. (eds.) Information Security. Lecture Notes in Computer Science, vol. 6531, pp. 346–360. Springer, Heidelberg (2011)
9. Dietz, M., Shekhar, S., Pisetsky, Y., Shu, A., Wallach, D.S.: Quire: lightweight provenance for smart phone operating systems. In: USENIX Security (2011)
10. Enck, W., Ongtang, M., McDaniel, P.: On lightweight mobile phone application certification. In: 16th ACM CCS, pp. 235–245. ACM (2009)
11. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: 18th ACM CCS, pp. 627–638. ACM (2011)
12. Felt, A.P., Wang, H.J., Moshchuk, A., Hanna, S., Chin, E.: Permission re-delegation: attacks and defenses. In: USENIX Security Symposium (2011)
13. Fuchs, A.P., Chaudhuri, A., Foster, J.S.: Scandroid: automated security certification of android applications. University of Maryland, Manuscript (2009)
14. Google: Dashboard, March 2014. http://developer.android.com/about/dashboards/index.html?utm_source=ausdroid.net#Platform
15. Grace, M., Zhou, Y., Wang, Z., Jiang, X.: Systematic detection of capability leaks in stock android smartphones. In: 19th NDSS (2012)
16. Nauman, M., Khan, S., Zhang, X.: Apex: extending android permission model and enforcement with user-defined runtime constraints. In: 5th ACM CCS (2010)
17. NC State University: security alert: New sophisticated android malware droidkungfu found in alternative chinese app markets (2011). http://www.csc.ncsu.edu/faculty/jiang/DroidKungFu.html
18. NIST: Cve-2013-4787 (2013). http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-4787
19. Ongtang, M., McLaughlin, S., Enck, W., McDaniel, P.: Semantically rich application-centric security in android. Secur. Commun. Netw. **5**(6), 658–673 (2012)
20. Smalley, S., Craig, R.: Security enhanced (se) android: bringing flexible MAC to android. In: NDSS (2013)
21. Squad, A.S.: Bug 9695860 (2013). http://blog.sina.com.cn/s/blog_be6dacae0101bksm.html
22. viaForensics: Defeating SEAndroid C DEFCON 21 Presentation. https://viaforensics.com/mobile-security/implementing-seandroid-defcon-21-presentation.html. Accessed August 3, 2013
23. Xiaoyong, Z., Yeonjoon, L., Nan, Z., Muhammad, N., XiaoFeng, W.: The peril of fragmentation: security hazards in android device driver customizations. In: 35th IEEE Security and Privacy, pp. 1–18. IEEE (2014)
24. Zhou, Y., Jiang, X.: Dissecting android malware: characterization and evolution. In: Security and Privacy (SP), pp. 95–109. IEEE (2012)
25. Zhou, Y., Zhang, X., Jiang, X., Freeh, V.W.: Taming information-stealing smartphone applications (on android). In: McCune, J.M., Balacheff, B., Perrig, A., Sadeghi, A.-R., Sasse, A., Beres, Y. (eds.) Trust 2011. LNCS, vol. 6740, pp. 93–107. Springer, Heidelberg (2011)