Function Escalation Attack

Chen Cao(✉), Yuqing Zhang, Qixu Liu, and Kai Wang

National Computer Network Intrusion Protection Center,
University of Chinese Academy of Sciences, Beijing, China
{caochen11,wangkai212}@mails.ucas.ac.cn,
{zhangyq,liuqixu}@ucas.ac.cn

Abstract. The prevalence of smartphone makes it more important in people's business and personal life which also helps it to be a target of the malware. In this paper, we introduce a new kind of attack called Function Escalation Attack which obtains functions locally or remotely. We present three threat models: Steganography, Collusion Attack and Code Abusing. A vulnerability in Android filesystem which is used in code abusing threat model is exposed as well. Three proof-of-concept malicious apps are implemented for each threat model. They could bypass static analysis and dynamic analysis. The result shows that function escalation attack could successfully perform malicious tasks such as taking pictures, recording audio and so on.

Keywords: Android security · Dynamic code loading · Function escalation attack · Vulnerability

1 Introduction

Modern smartphone is ubiquitous nowadays. It allows users to install apps from online application(abbr. app) markets to enforce the smartphone's capabilities. As a result, it plays an important part in people's business and personal life which helps it to be a target of the malware writers.

This work focuses on the Android platform which is one of the most popular mobile operating systems. Android OS is a permission-based framework that each app must apply for necessary permissions to process its action. Furthermore, according to the OS, each app is an individual user which means they can't access each other's resources without appropriate permissions. However, this framework is vulnerable to confused deputy and collusion attacks, etc. Although several tools have been proposed to solve these problems, such as TaintDroid [1], XmanDroid [2], CHEX [3], SCANDAL [4] etc. these tools are just mitigating the damage and they can't provide an absolute solution for these attacks.

In this paper, we demonstrate a new kind of attack which could bypass the static and dynamic analysis for the apps. We name it Function Escalation Attack. Function Escalation Attack is a kind of attack that malware application doesn't have any malicious behavior at first, but it could obtain these behaviors locally

J. Tian et al. (Eds.): SecureComm 2014, Part I, LNICST 152, pp. 481–497, 2015.
DOI: 10.1007/978-3-319-23829-6_33

or remotely after being installed. Function Escalation Attack includes update-attack studied in [5]. Update-attack gains behaviors remotely. Once installed, the malicious app would trick the user to download the newest version from the Internet and to install the newest one. In fact, the newest version has the malicious codes. For Android, the newest version could be an intact APK or DEX file or the library in *.so file, malformed file like .mp3 or something leveraging the vulnerabilities in the system. However, this method could be prevented by the Internet stream monitor [5]. In addition, Google has banned self-updating Android apps in Google Market [6]. Each app could only be updated via Google. This means that Google would review the updating parts to prohibit the aforementioned attacks.

The attack depicted in this paper uses little Internet stream or none. Attackers can release a benign-looking application without any harmful functions originally using this method. After being installed the app could acquire the functions locally and perform malicious tasks.

The key idea behind this kind of attack is that, instead of obtaining the vicious parts from the Internet, the malware can gain these from the resource files, the APK files of the collusion evil apps or the benign apps with the use of a vulnerability in Android. The vulnerability would be described in Sect. 3.3.

Our work consists of three threat models of function escalation attack: Steganography, Collusion Attack and Code Abusing. We have implemented three proof-of-concept malicious apps for each threat model. The result showed that we could successfully gain the functions from other benign apps and perform malicious tasks, such as taking pictures and recording audio. In summary, the contributions of this paper are the following:

- We propose a new kind of attack called function escalation attack with three threat models that the original app has no malicious functions in it but could get the functions after being installed with the use of little Internet stream or none.
- We are the first to reveal this vulnerability in Android that the bad policies used in Android file system leads to that the APK files are exposed which could be successfully abused.
- Three proof-of-concept apps are implemented on mobile devices running Android. Defense methods are discussed.

The paper is organized as follows. Section 2 gives background on Android security model. Java class loader and reflection are also described there. Section 3 presents the design and implementation of these three threat models. In Sect. 4, two case studies are presented. Section 5 discusses the improvement of our attack and the countermeasures against this attack. Section 6 presents related work. The conclusion is in Sect. 7.

2 Background

Android is a mobile platform which implements the software stack on Linux. This stack includes applications, application framework and libraries from top

to bottom. Android application are written in Java and C/C++. Android has its own Java virtual machine called Dalvik VM and each app runs within its own Dalvik VM instance. In this section, Android security model and Java class loader mechanism in Android are introduced. Java reflection is also depicted.

2.1 Android Security

In the following we briefly introduce the core security mechanisms of Android: (a) Sandbox (b) Application permissions (c) Application sign and (d) File system permissions. More details can be found in [7,8].

Sandbox: The data and code execution of each application in Android system is isolated from each other which means that an app runs in its own sandbox, including its native code. So if one app is exploited, the attacker could just be inside the app's sandbox without any damage to other apps. According to the OS level, each app is a standalone user with a unique UID(User ID) that has limited permissions. By default, apps cannot interact with each other without proper permissions. Furthermore, two applications' sandboxes could be merged together only under the harsh condition that they have the shareUID in the manifest XML file and the same signature. Beyond that, applications cannot access others' data and code.

Application Permissions: An Android application could only access a limited system resources. If an app wants more resources, it has to request more permissions. Permissions have four levels: normal, dangerous, signature and signature-or-system. Details could be referred in [9].

Application Sign: Each application should be signed by the developers with their self-certified key. This mechanism doesn't provide any protection against malware apps but can hold the developers' identity. As mentioned above, applications should have the same signature to be put into the same sandbox with the same UID.

File System Permissions: Android is a Linux-based system, so it has the same file system permissions as Linux, that a user can't access other users' files casually. Generally speaking, a file has three permission types (Read, Write and Execute) for a user, a group and other users. This kind of access policy is discretionary access control (DAC). Android has been reinforced by SELinux which provides mandatory access control policy (MAC) since version 4.3.

2.2 Java Class Loader and Reflection

Java Class Loader: A class loader is an object that is responsible for loading classes. In Android, there are two main classes for this purpose: DexClassLoader and PathClassLoader.

DexClassLoader can load classes from JAR and APK files containing a classes.dex entry. When initializing a DexClassLoader instance, developers must specify the file names. Besides, the path of the optimized DEX file, namely ODEX file, should be specified. For the sake of security, the ODEX file's output path ought to be a private space that cannot be overwritten by other apps [10].

PathClassLoader is slightly different from DexClassLoader for it just operates on a list of files and directories in the local file system and cannot attempt to load classes from the network. Android employs this class loader for the system class loader and for its application class loaders.

Java Reflection: Java reflection makes Java programs so flexible that they don't need to know the names of the classes, methods etc. at compile time. It can be used for observing and modifying program execution at runtime. When Java program uses Java reflection, it is hard for the static program analysis tool to get the semantics of this program. Namely it is an important open problem in Android.

2.3 APK File Format and Android Parsing Method

Android application package file (APK) is the package file format used to distribute and install application software and middleware onto Google's Android operating system [11]. This kind of file format is actually Zip file format. Therefore, it shares the same characteristics as Zip file format. Figure 1 depict APK file format.

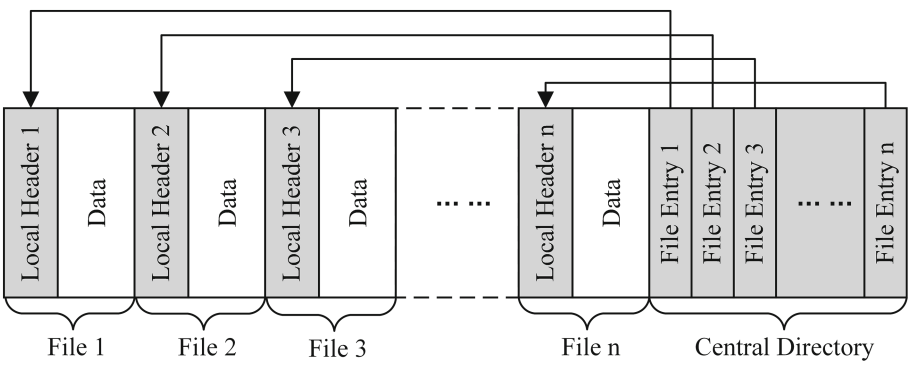

Fig. 1. APK file format

According to Android's parsing methods, APK file's central directory is firstly acquired. After the central directory is extracted from the file, each file entry's offset is parsed subsequently. Thus, the file entry is obtained according to the offset in the central directory. In other words, this offset could be modified and some other things could be inserted between two file entries.

2.4 Lua

Lua, a powerful, fast, lightweight, embeddable scripting language, has been used in many industrial applications, with emphasis on embedded systems and games [12]. Lua is the fastest language in the realm of interpreted scripting languages according to several benchmarks. Lua is a powerful but simple language which provides meta-mechanisms for implementing features, instead of a host of features directly in the language. Lua's interpreter is always small.

LuaJ is a Lua interpreter based on the 5.2.x version of Lua implemented in Java [13]. Contrasting with other Lua interpreters implemented in Java, LuaJ shows a better performance. Besides, LuaJ is small-size and less than 400KB after being compiled.

3 Design and Implementation

Before introducing the design and implementation of the three threat models, we first discuss the overview of function escalation attack. Then three threat models would be depicted. Each model has its own characteristics with different techniques. The limitation of the three threat models is given at the end of this section.

3.1 Overview

The key point of function escalation attack is how to obtain the functions which are not in the app originally after the app has been installed in the Android system. The high level idea of our attack is very intuitive. The malicious codes that a malware could gain could only come from two ways, namely itself or other apps. The other apps could be collusive or benign. Therefore, our three threat models of function escalation attack are the three ways to acquire functions. Figure 2 depicts the relationship of three threat models and function escalation attack.

3.2 Steganography

Steganography is a technique of hiding confidential information within any media [14]. In this scenario, the malicious code could be stored into the media files such as pictures and audios. Specifically, PNG and Zip files are used to illustrate this method in Android.

PNG, short for Portable Network Graphics, is a raster graphics file format that supports lossless data compression. Details could be found in RFC 2083 [15]. A PNG file's magic number is '89504E470D0A1A0A' followed by chunks. A chunk includes four parts: length (four bytes), chunk type/name (four bytes), chunk data (length bytes) and CRC (cyclic redundancy code/checksum; four bytes). The picture viewer would ignore the chunks that fail to meet the specification. In this way, the malicious codes could be inserted there and

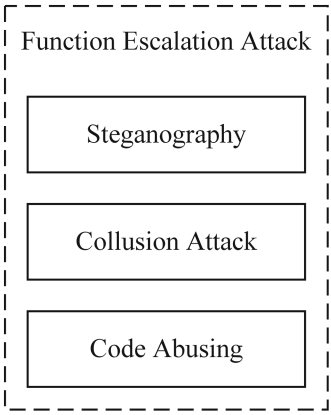

Fig. 2. Overview of function escalation attack

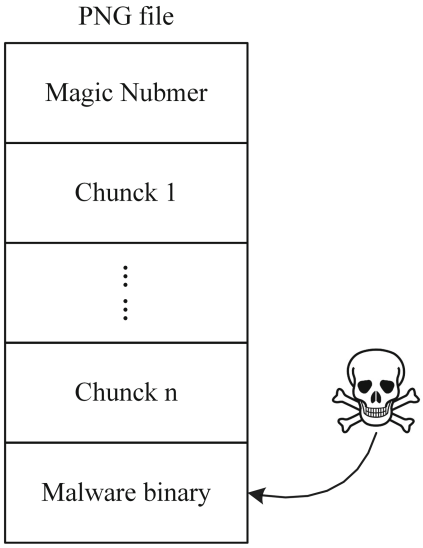

Fig. 3. Insert malicious code into PNG file

extracted after the app is installed. Figure 3 illustrates the PNG file structure where the malicious code could be placed.

Zip is an archive file format. As depicted in Sect. 2.3, Zip file could be inserted into anything between file entries. After that, the offset in central directory should be modified to fit the changes. Figure 4 shows the result of the process.

In Android system, apps often put the media files into the asset directory which is easy to access. In order to achieve a better performance, malicious code's extracting function is implemented in the native code in a *.so file. When the app has been installed, the code would be extracted into the app's private directories

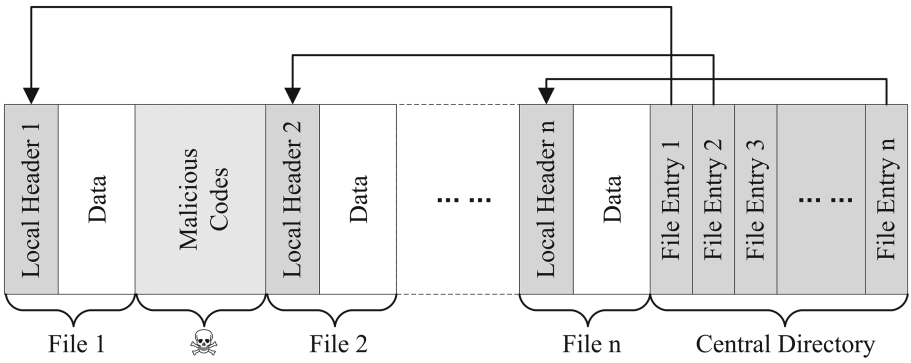

Fig. 4. Insert malicious code into Zip file

at a proper time. Some more covert measures can be done: (*a*) Partition an entire APK file into small parts. (*b*) Use stream cipher to encrypt the divided binary parts to make them look like the carrier's content.

After being installed into the system, the small parts would be extracted from the PNG files, decrypted and assembled together in the app's own private directory.

3.3 Collusion Attack

Collusion attack means that except malicious app A, there is also a malicious app B which provides the resources that app A needs. The resources could be B's codes or B's media files where the malicious code is hidden. Moreover, app A and app B can store different parts of the malicious code that even if some malicious code has been analyzed, the other parts are also hidden. So the whole picture is unbeknown. As app B can expose anything it has to shared place such as SD card, this kind of attack is easier to realize but harder to detect. Figure 5 shows this process.

We implement two apps, namely app A and app B, serving as the invoking app and the collusion app respectively. After being installed into the system, app A would scan the system to find app B. If B is not found, A would do nothing but recommend app B to the user. If B has been installed, A would look for the unique directory in the SD card which stores the binary file extracted from B. Having found the binary file, A copies the file into its own directory, deletes the directory and sends an intent to B to tell it that A has acquired the binary file. Then B would never extract the binary file again. The above mentioned method uses file system as a kind of covert channel. More details about overt and covert channel could be found in [16]. To be more elusive, we also separate the binary file into several parts and store them in app A and B. After obtaining the different parts in B, A would firstly assemble the binary file as above.

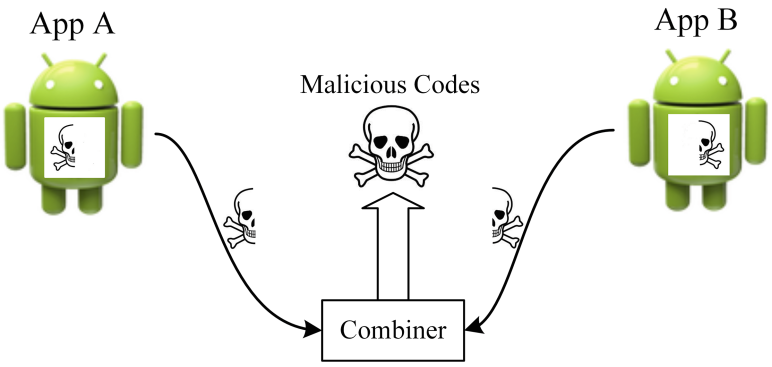

Fig. 5. Collusion attack

3.4 Code Abusing

As mentioned in Sect. 1, Android has a vulnerability that could be used to carry out an attack. The vulnerability is inappropriate access permission in the file system. The directory "/system" could be read by anyone. In addition to that, although the directory "/data/app" could be read by no one, the APK files in this directory could be accessed by anyone. This inappropriate policy could be abused. Any app could load the APK file and invoke the methods without any restrictions.

To be more malicious, little Internet stream is needed. After being installed, the malicious app could firstly gather the installed apps' information and then send them to the server. The information of one app consists of three parts: the app's name, version and APK hash value. The three parts are used for the server to gain the exact app packages. After having obtained the app package, the server would analyze it and then extract the methods that could be invoked by the malicious app. Then the server implements the invoking Lua script and sends it back to the malicious app in the target Android system. At present, the scripts are all implemented manually. The malicious app has a Lua interpreter, i.e. LuaJ, which executes all the scripts from the server. The details are illustrated in Fig. 6.

Reverse Engineering. After acquiring the exact target app APK file, the server needs to analyze it to gain the invoking methods. Although the reverse engineering of the Android application is a mature technique, how to locate the target function path is a problem. At present, our methods are all manual: (a) Use android-apktool [17], a tool for reverse engineering Android APK files, to decompile the apps. (b) Use dexdeps [18], a tool that could dump a list of fields and methods that a DEX file uses but does not define. (c) Locate the exact methods and fields in the app's smali source file with the permissions from the AndroidManifest.xml file and the list of fields and methods gained by dexdeps. (d) The last step is empirical, i.e. locate the function paths.

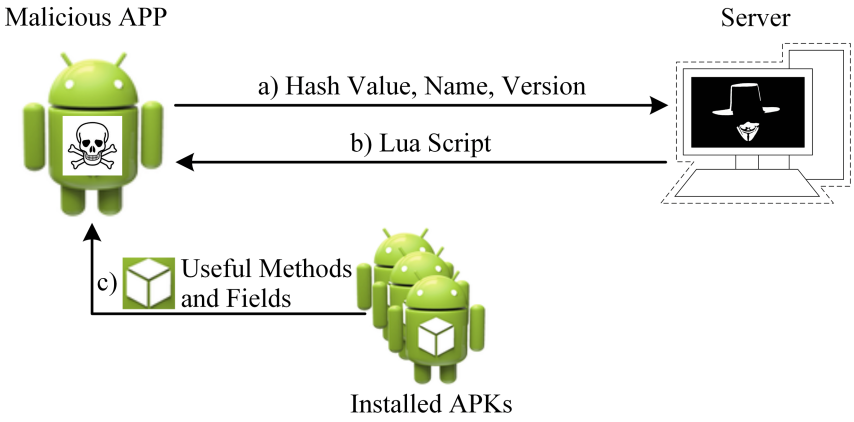

Fig. 6. (a) Obtain the app's three parts and send them to the sever; (b) The server pushes the Lua script; (c) Invoke the target app's methods

Lua and LuaJ. LuaJ is a small Lua interpreter implemented in Java. The malicious app contains a modified LuaJ. As the original LuaJ cannot use Java reflection to invoke the arbitrary APK file, it should be modified to have this ability. The architecture of the invoking framework is depicted in Fig. 7. Listing 1 is a short example of the Lua script for the malicious app to invoke the target APK file.

Listing 1. Lua Script example to invoke the target app's methods to gain the contact

```
1  -- initiate context
2  luaContext = context:getApplicationContext()
3
4  -- bind class loader
5  classLoader = luajava.bindClass("java.lang.ClassLoader")
6  cl = classLoader:getSystemClassLoader()
7
8  -- initiate the arguments
9  libpath = "/data/data/com.baidu.netdist/lib"
10 dexpath = "/data/app/com.baidu.netdisk-2.apk"
11 dexoutputpath = "/data/data/com.example.caochen/app_dex"
12 cr = luaContext:getContentResolver()
13
14 -- create an instance of DexClassLoader
15 dcl = luajava.newInstance("dalvik.system.DexClassLoader",dexpath,
       dexoutputpath,libpath,cl)
16 -- load target class
17 readClass = dcl:loadClass ("com.baidu.pimcontact.contact.dao.contact.read
       .ContactReadDao")
18 -- create an instance of class ContactReadDao
```

```
objRead = luajava.new(readClass,cr)
-- get all the IDs of the contact
idList = objRead:getAllRawId()
-- get the contact list
contactList = objRead:getInfoByIds(idList)

contactClass = dcl:loadClass("com.baidu.pimcontact.contact.bean.
    contacts.Contact")
objcontact = luajava.new(contactClass)

-- get a piece of contact information
res = objcontact:build(contactList:get(1))

return res
```

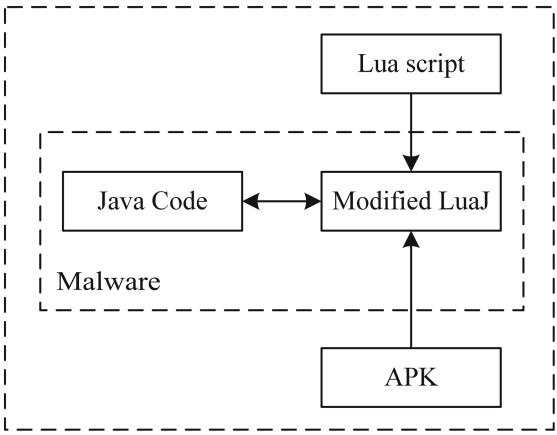

Fig. 7. Architecture of the invoking framework

3.5 Limitation

Function escalation attacks heavily depends on Java Class Loader and Java Reflection. Therefore, the limitation of Java Class Loader and Java Reflection influences the attack. For example, DexClassLoader and PathClassLoader should be used in Android API level 14 and above, i.e., Android 4.0 and above. It imposes restriction on usable range of function escalation attack. However, the devices of Android 4.0 and above occupy the most market share [19].

Steganography's limitation is the host's size. As the PNG picture is not often large, the size of hidden code is restricted. As for the Zip file, there is no limitation.

Collusion attack needs two or more apps to be installed in the system, which is usually hard to achieve in practice.

In theory, the third threat model could leverage any codes in the target app. However, if the target app contains some hard-coded values in the target methods, this threat model can do nothing. For example, BaiduMap [20], one of the most popular map applications in China, has the permission to send SMS. But the destination number is hard-coded. So it could not be abused to send SMS.

Furthermore, code obfuscation techniques may increase the difficulty of analyzing apps. Moving key algorithms or secrets into native code also has the similar effect.

4 Case Study

In this section, we depict two case studies. The first one is to take a picture through YouDao dictionary application [21] which is the most popular dictionary app in China. The second one is to eavesdrop through YouDao cloud note application [22] which is the most popular note app in China.

4.1 Photograph

The version of YouDao dictionary app is 4.2.2 which was the newest as we wrote this paper. This app has the function of optical character recognition (OCR). So it has the permission to take pictures, i.e. Camera. Hence, what we want to get from it is its camera function. We firstly analyze this app and get the target class "CameraManager" that should be loaded. Then the starting and stopping path is shown in Fig. 8. The starting path is the list of methods that should be reflected to take a photo. The stopping path is the list of methods that should be reflected to stop the camera.

In the proof-of-concept malicious app, we just take a picture when the surfaceview is first created. So We invoke method takePhoto() in surfaceCreated of SurfaceHolder.Callback.

4.2 Eavesdrop

The version of YouDao cloud note app is 3.5.0 which was the newest as we wrote this paper. This app could make the audio part of the notes, so it has the function of recording audio. We firstly analyze this app and get the target class "YNoteRecorder" that should be loaded. This app uses native code to enhance its audio record, so its native code should be loaded together. To our surprise, the native code in the directory "/data/data/(apps name)/lib" could be read by everyone. This makes it easier to invoke the target app without extracting the library from its APK file first. In this case, we use its static method "loadLibSuccess" to ensure the success of loading the native code. Then starting path is as follows: YNoteRecorder() $\Longrightarrow$ start(). The stopping path is as follows: stopRecord().

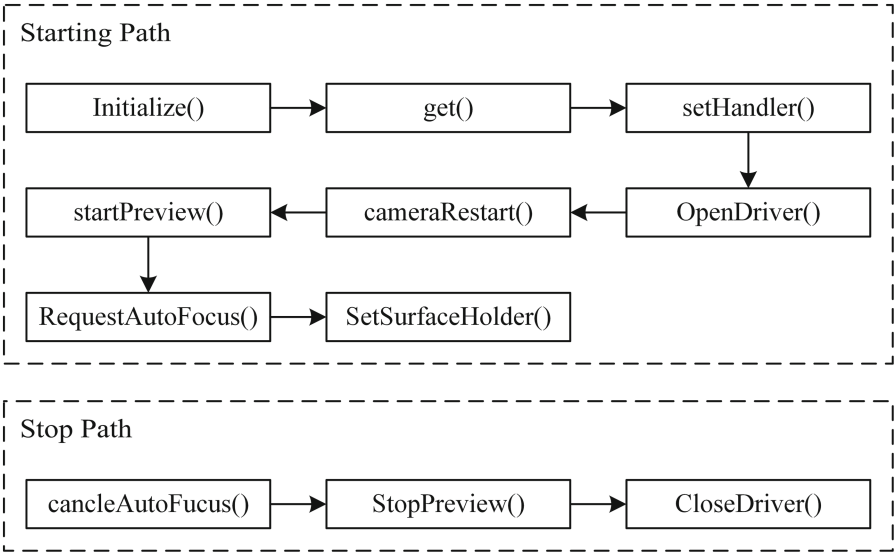

Fig. 8. Staring and stopping path of taking photo

5 Discussion

5.1 Improvement to the Attack

We believe the attack could be improved by the following measures.

Performance. As we analyzed lots of apps, we found that some apps could not be invoked appropriately. For example, BaiduMap [20] put its SMS destination number in hard code and a new number cannot be put in. So if the malware app could copy the target app out, modify the target binary app and then load the modified app, the performance would be better.

Universality. The attack we implemented in this paper just leverages the third-part apps as Android puts third-party apps' DEX file into the APK file. However, native apps' DEX files are outside its APK file and have been optimized to ODEX files. If the malware app could translate the ODEX files into DEX files and put them back into their APK files, function escalation attack would have a wider use.

Stealthiness. The attack could bypass static analysis as Java reflection is an open problem for it. At present, function escalation attack uses covert channel to translate privacy or secret data to bypass some dynamic analysis. However, when using dynamic analysis against it, the attack could be weak at the stealth of gaining data. To further reduce detectability, the attack could take advantage of

anti-detection techniques to detect the environment. If the malware has detected that the environment is not the common environment in a mobile device, it would not perform malicious tasks.

5.2 Defense

The main focus of this attack is how to gain function to archive malicious behavior. In this section, several security extensions for Android are discussed. These include dynamic analysis, static analysis and system enforcement. The last are our proposed methods.

Kirin is an extension to Androids application installer [23,24]. It checks apps permissions at install-time and denies the app that has the permission set violating a given system policy. As the malware app just invokes other apps function, the permissions used to do the function should be applied first. Kirin more or less could detect the permission violating. However, function escalation attack could use covert channel without applying any permissions violating policies.

TaintDroid is an information flow tracking system which tracks sensitive data in Android against privacy leakage [1]. It utilizes dynamic taint analysis within Dalvik VM interpreter. TaintDroid mainly addresses the data flows, whereas tracking the control flow will likely result in much higher performance penalties. As a result, taintdroid cannot detect some behavior of this kind of attack.

VetDroid is a dynamic analysis platform for reconstructing sensitive behaviors in Android apps from a novel permission use perspective [25]. It combined dynamically tracing of the permission requests for resources usage by applications, with tracking sensitive operations on the granted resources (using taint tracking). This combination enables researchers to understand how applications utilize the permissions to access sensitive system resources. Our attack must apply for the permissions to perform the malicious tasks. However, VetDroid is an off-line analysis platform. If the malicious could detect the environment and does nothing vicious in the virtual machine, VetDroid cannot figure out whether the app is malicious. Namely, the malware pretends to be benign when the environment is abnormal.

CHEX is a static analysis method to automatically vet Android apps for component hijacking vulnerabilities such as permission leakage, unauthorized data access, intent spoofing, and etc. [3]. As we have described above, function escalation attack is a kind of attack gaining functions dynamically. Static analysis method could find nothing from the originally app.

SELinux on Android is used to apply control policies. It helps to control access to application data and system logs, reduce the effects of malicious software, and protect users from potential flaws in code on mobile devices [26].

SELinux is reinforced in Android from 4.3. However, as Android 4.4 has been released, we find that the file system permission vulnerability still exists. This means our function escalation attack could be carried out in this newest Android version.

Our Method: If the APK files' access policy is changed so that others couldn't read them, all the resources and assets would not be accessed by the system. Namely that's not an option. As described above, function escalation attack heavily depends on Java reflection. Thus, the class loader such as DexClassLoader and PathClassLoader is crucial. So, our method is to monitor the class loader and to get the loaded codes in the real mobile devices. The acquired codes would be pushed into the server to be analyzed and the user could know the result from the server. This method depends on network heavily. However, it could detect all the loading process and nested class loading.

6 Related Work

Reflection in Android. [27] and [28] have introduced a malware app in the wild which uses reflection in its code. This app dynamically loads its child package 'anserveb.db' from the assets directory which is actually an external malicious package. It is like our first threat model that hiding malicious codes in the resource files. [29] analyzed unsafe and malicious dynamic code loading in Android applications. It gave a kind of attack that loaded the malicious code from the internet which could be prevented by [5] and is not in our scope. It systematically analyzed the security implications of the ability to load additional code in Android. However, it missed our attack.

Privilege-Escalation Attack. Android, a permission-based framework, is shown to be vulnerable to application-level privilege escalation attacks [30–32]. A malicious app could escalate granted permissions and bypass restrictions imposed by its sandboxes. Our attack does not escalate the malware's permissions at runtime but obtain functions dynamically.

Component Hijacking. Android's apps are component-based. Component hijacking let an unauthorized app read and write data originally inside other apps through their components [3]. This attack is similar with our attack. However, our attack doesn't depend on whether the target app's component is exposed.

Sensor-Based Attack. Soundcomber [32], Accessory [33], TapLogger [34] and PlaceRainder [35] are all sensor-based malware applications. Soundcomber is a trojan with few and innocuous permissions, that can extract a small amount of targeted private information from the audio sensor of the phone. It could steal

sensitive data such as credit card and PIN number from both tone-and speech-based interaction with phone menu systems using targeted profiles for context-aware analysis. Accessory is password inference application using accelerometers. This app has two collection modes: areas and character. After collecting the accelerometer data, the app would do some data analysis which infers password in the end. TapLogger is a trojan application inferring a user's tap input to a smartphone with its integrated motion sensors. This trojan application must learn the motion change pattern of tap events when the user is interacting with it. Then it could infer the user's sensitive inputs with the pattern. PlaceRainder is a trojan application through completely opportunistic use of the camera that the attacker could reconstruct rick three-dimensional models of the smartphone owner's personal indoor spaces. Our function escalation attack do not use sensor necessarily to archive malicious behavior. However, it could leverage the sensor-based apps to improve the attack.

Control Flow Change. JekyII on iOS [36] is a malware application on iOS which could defeat Apple's mandatory app review and code signing mechanisms. The key idea of it is to make the apps remotely exploitable and subsequently introduce malicious control flows by rearranging signed code. Our function escalation attack is also changing control flow by invoking the dynamic loading classes which could bypass the static analysis for the app.

7 Conclusion

This paper introduces a new kind of attack – Function Escalation Attack in Android. A malicious application doesn't have any malicious behavior at first, but it could obtain these behaviors locally or remotely after being installed. Three detailed threat models are presented: Steganography, Collusion Attack and Code Abusing. Steganography is a technique of hiding confidential information within the malware self. Collusion attack means that except malicious app A, there is also a malicious app B which provides the resources that app A needs. Code abusing leverage a vulnerability in Android that the APK files could be accessed by anyone, which means their codes and resources can be used by Java reflection. Defense methods, such as dynamic analysis, static analysis and system enforcement, are discussed. A new defense method is also proposed.

Acknowledgement. This research is supported by National Natural Science Foundation of China [61272481].

References

1. Enck, W., Gilbert, P., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation. OSDI 2010, pp. 1–6. USENIX Association, Berkeley (2010)

2. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., d Sadeghi, A.R.: Xmandroid: a new android evolution to mitigate privilege escalation attacks. Technical report TR-2011-04, Technische Universität Darmstadt (2011)
3. Lu, L., Li, Z., Wu, Z., Lee, W., Jiang, G.: Chex: statically vetting android apps for component hijacking vulnerabilities. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 229–240. ACM (2012)
4. Kim, J., Yoon, Y., Yi, K., Shin, J., Center, S.: Scandal: static analyzer for detecting privacy leaks in android applications. In: MoST (2012)
5. Tenenboim-Chekina, L., Barad, O., Shabtai, A., Mimran, D., Rokach, L., Shapira, B., Elovici, Y.: Detecting application update attack on mobile devices through network features. In: The 32nd IEEE International Conference on Computer Communications (2013)
6. Google Play Developer Program Policies. https://play.google.com/about/developer-content-policy.html. Accessed March 2014
7. Android Security Overview. http://source.android.com/devices/tech/security/index.html. Accessed March 2014
8. Enck, W., Ongtang, M., McDaniel, P.D., et al.: Understanding android security. IEEE Secur. Priv. **7**(1), 50–57 (2009)
9. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, pp. 627–638. ACM (2011)
10. Dexclassloader. http://developer.android.com/reference/dalvik/system/DexClassLoader.html. Accessed March 2014
11. Google, G.S.: Inside the android application framework (2008). https://sites.google.com/site/io/inside-the-android-application-framework
12. Lua. http://www.lua.org/about.html
13. LuaJ. http://luaj.org/luaj/README.html
14. Petitcolas, F.A., Anderson, R.J., Kuhn, M.G.: Information hiding-a survey. Proc. IEEE **87**(7), 1062–1078 (1999)
15. Rfc 2083. http://tools.ietf.org/html/rfc2083
16. Marforio, C., Ritzdorf, H., Francillon, A., Capkun, S.: Analysis of the communication between colluding applications on modern smartphones. In: Proceedings of the 28th Annual Computer Security Applications Conference, pp. 51–60. ACM (2012)
17. Android-apktool. https://code.google.com/p/android-apktool/
18. Dexdeps. https://android.googlesource.com/platform/dalvik.git/+/android-4.2.2_r1/tools/dexdeps
19. Google, I.: Android market share. http://developer.android.com/about/dashboards/index.html. Accessed March 2014
20. Baidu Map. http://map.baidu.com/
21. Youdao Dictionary. http://cidian.youdao.com/mobile.html
22. Youdao Clound Note. https://note.youdao.com/index.html
23. Enck, W., Ongtang, M., McDaniel, P.: Mitigating android software misuse before it happens. Technical Report NAS-TR-0094-2008, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA (2008)
24. Enck, W., Ongtang, M., McDaniel, P.: On lightweight mobile phone application certification. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, pp. 235–245. ACM (2009)

25. Zhang, Y., Yang, M., Xu, B., Yang, Z., Gu, G., Ning, P., Wang, X.S., et al.: Vetting undesirable behaviors in android apps with permission use analysis. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, pp. 611–622. ACM (2013)
26. Shabtai, A., Fledel, Y., Elovici, Y.: Securing android-powered mobile devices using selinux. IEEE Secur. Priv. **8**(3), 36–44 (2010)
27. An analysis of the anserverbot trojan. http://www.csc.ncsu.edu/faculty/jiang/pubs/AnserverBot_Analysis.pdf
28. Grace, M., Zhou, Y., Zhang, Q., Zou, S., Jiang, X.: Riskranker: scalable and accurate zero-day android malware detection. In: Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services, pp. 281–294. ACM (2012)
29. Poeplau, S., Fratantonio, Y., Bianchi, A., Kruegel, C., Vigna, G.: Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In: NDSS, vol. 14, pp. 23–26 (2014)
30. Felt, A.P., Wang, H.J., Moshchuk, A., Hanna, S., Chin, E.: Permission re-delegation: attacks and defenses. In: USENIX Security Symposium (2011)
31. Davi, L., Dmitrienko, A., Sadeghi, A.-R., Winandy, M.: Privilege escalation attacks on android. In: Burmester, M., Tsudik, G., Magliveras, S., Ilić, I. (eds.) ISC 2010. LNCS, vol. 6531, pp. 346–360. Springer, Heidelberg (2011)
32. Schlegel, R., Zhang, K., Zhou, X.Y., Intwala, M., Kapadia, A., Wang, X.: Soundcomber: a stealthy and context-aware sound trojan for smartphones. In: NDSS, vol. 11, pp. 17–33 (2011)
33. Owusu, E., Han, J., Das, S., Perrig, A., Zhang, J.: Accessory: password inference using accelerometers on smartphones. In: Proceedings of the Twelfth Workshop on Mobile Computing Systems and Applications, p. 9. ACM (2012)
34. Xu, Z., Bai, K., Zhu, S.: Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors. In: Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks, pp. 113–124. ACM (2012)
35. Templeman, R., Rahman, Z., Crandall, D., Kapadia, A.: Placeraider: virtual theft in physical spaces with smartphones (2012). arXiv preprint, arXiv:1209.5982
36. Wang, T., Lu, K., Lu, L., Chung, S., Lee, W.: Jekyll on ios: when benign apps become evil. In: Presented as Part of the 22nd USENIX Security Symposium (USENIX), pp. 559–572 (2013)