Armor: Android Security Based Tools

Aniket Thigale, Juhi Yardi, Samkit Shah, Rutugandha Bapat
Pune Institute of Computer Technology

Abstract

Android, although being based on a Linux kernel, has specific limitations due to its deployment in a constrained environment of mobile phones thus making it harder to detect and react upon malware attacks. In addition to this, Android being open source, thousands of applications are being developed and installed every day thus making it more vulnerable to data leaks. Hence, Android is in need of a security tool which strengthens the user privacy and also safeguards the Android kernel. We propose a system which handles threats at two levels which are application level & kernel level. At the application level our system will monitor the installation and updation phase of an application. The system also will monitor the real-time communication between android applications and verify the inter-process communication against a set of pre-defined security policies. The goal is to prevent malicious applications from exploiting transitive permission properties to enable privilege escalation. At the kernel level we will perform dynamic behavioral analysis by hijacking the system calls made by the application running instances on the device. Hence our system acts as armor against the security threats to the Android based smart devices.

Problem Statement

Developing a system for the Android Platform that monitors the life cycle of each application for any unauthorized activity, intent and malicious behavior using static and dynamic analysis and notify the user & perform necessary actions.

I. Execution Plan

We have used client-server architecture for the implementation of our system. The client is the Android device which has the service installed while the server is a Linux-x64 based system running Ubuntu 12.10. Our service starts on the Android device when it boots and runs continuously in the background thereafter. In case of a new application download and installation, it extracts the permissions from AndroidManifest.xml required by the application and prompts the user for entering the category to which the application belongs to. It queries the server for the authorized permissions required by that particular category. The database at the server contains a mapping of category and its most commonly used safe permissions which have been determined by a study conducted by us after examining 50 common applications. A cache also stores the permissions of the three most recently checked categories. So in case of a miss, the server gets queried. These permissions are then compared and the user is notified about the extra permissions that the application requires and is given the choice to either continue or uninstall the application. These permissions are first encoded in the form of bit stream so as to increase the speed of the entire process. The next part of our system involves a Loadable Kernel Module which catches all the system calls made by an application and logs them. We have used Android 4.0.3_r1 platform source code downloaded from the Android Development site and the goldfish kernel source for building the kernel for the emulator.

II. Capability Leak Detection and Transitive Privilege Escalation

Android depends on privilege separation to isolate applications from each other and from the system. However, a recent research reported that a genuine application exploited at runtime or a malicious application can escalate granted permissions. The attack depends on a carelessly designed application which fails to protect the permissions granted to it. In this research, we propose a vulnerability checking system to check if an application can be potentially leveraged by an attacker to launch such privilege escalation attack. Basically, Android is considered a privilege-separated operating system. It means that every application runs in its own Davik virtual machine and each of them is assigned a distinct system identity. The purpose of this mechanism is to isolate applications from each other as well as from the system. Unless otherwise requested, an application is not allowed to perform any operations that would adversely impact other applications, the operating system, or the user. If an application needs to obtain extra capabilities, it needs to declare all of the permissions that it needs in its AndroidManifest.xml. At application installation time, the user decides whether to grant the permissions to the application or not. During the run-time of the application, no further checks with the user are done. The application either was granted a particular permission when installed and can use that feature as desired or the permission was not granted and any attempt to use that feature will fail without prompting the user. For instance, to be able to access the Internet, an application needs to have the INTERNET permission. However, recent research has showed that an application with less permissions (a non-privileged caller) is not restricted to access components of a more privileged application (a privileged callee). Such attack is called privilege escalation attack or confused deputy attack. It essentially allows a potentially malicious application to indirectly obtain extra capabilities leaked from a benign application which did not do a good job to protect the permissions granted to it. In order to prevent this sort of attack, it is necessary for applications to enforce additional checks on permissions to ensure that the callers have the required permissions before doing any dangerous actions. Since most of the application developers are not security experts, delegating the task of performing these checks to them is an error prone approach. The aim of this research is to develop an automatic analysis system for detecting capability leaks in Android applications and find out how prevalent they are in existing applications. The resulting system is useful for developers to make sure that their applications, while not malicious on their own, do not leak capabilities to other applications. Moreover, Android users can use our system to search for capability leaks in an application before installing it. Our System first parses the AndroidManifest.xml file which clearly defines 2things. First, the permissions an application needs. Second, the permissions other applications need to have so as to interact with the components of that application. It is mandatory that every Android application includes the AndroidManifest.xml in its Android package (APK) file. Such a rule is enforced by the Android system. After that, it identifies components that are potential sources of capability leaks. For each of such components, it looks into its source code and uses inter procedural control flow searching to follow the taint propagation. It then obtains data paths that lead to capability leaks. It raises an alarm whenever such paths are found.

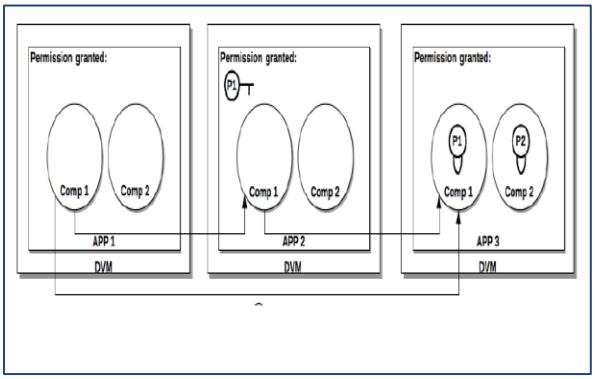


Fig 1.0. Privilege Escalation Attack

In this section, the privilege escalation (also known as confused deputy attack) on Android is

described in details. This attack was first found by Lucas Davi et al. in. The problem was stated as follows: An application with less permissions (a non-privileged caller) is not restricted to access components of a more privileged application (a privileged callee).An example of privilege escalation attack on Android is illustrated in the figure below .In the figure, three Android applications are running in their own DVMs. As shown in the figure, application 1 possesses no permissions. On the other hand, application 2 is not protected by any permissions. Therefore, other applications can interact with components of application 2. As a result, component 1 of application 2 can be accessed by both components of application 1. It can be known from the figure that application 2 possesses permission P1 (as shown by the key at the top left corner of application 2). Therefore, both components of application 2 can interact with component 1 of application 3 which is guarded by permission P1 (as shown by the keyhole on it). It is shown in the figure that component 1 of application 1 is interacting with component 1 of application. 2. However, it is not allowed to interact with component1 of application 3 as it does not own permission P1.As a result, component 1 of application 2 can interact with component 1 of application 3. Consequently, even though component 1 of application 1 is not permitted to access component 1 of application 3, it can access it by interacting with component 1 of application 2. In this scenario, the attack is successfully launched and the result of it is that privilege of application 2 (permission P1) is escalated to application 1. It is not difficult to prevent this attack. Component 1 of application 2 should state in its tag in the AndroidManifest.xml that components accessing it must own permission P1. The other option would be to use the checkPermission API call in its source code to check whether or not the component sending the intent message to it owns permission P1 and reject to access application 3 if it does not. In reality, most developers are not security experts. In other words, they may not notice the possibility of any capability leaks through their applications and the consequences of them. Even if they can notice that, they may not be motivated to take any measures to prevent such attacks as it is not the deputy itself that is harmed in the attack ultimately. However, the consequences that could be result from such attack should not be underestimate. They could be serious since Android provides a set of functionality rich API calls. They can be used to perform a lot of security sensitive operations which include getting current location, sending text messages, making phone calls, and reading information on NFC cards. Recently, android applications can even be used to manipulate various devices using the Android ADK framework. It is crucial to protect applications that are equipped with permissions to perform such dangerous actions. It is not hard to see that the possibility of such an attack hinges on the existent of a deputy (application 2 in our previous example). The reason is that it serves as the stepping stone for other applications to acquire extra capability. Such a deputy leaks to other applications its capabilities which are the permissions it owns and hence, the operations it can perform. The goal of this research is to identify vulnerabilities in Android applications that can lead to such capability leaks.

III. Our system Design

A. System overview

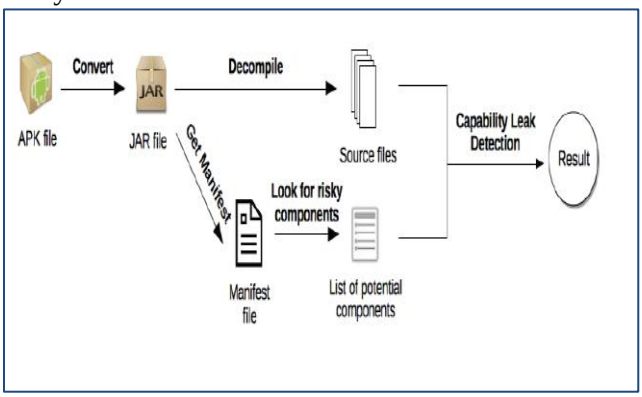


Figure 1.1 : System Overview

The APK file of the Android application to be analyzed is first converted into a JAR file. This is done by dex2jarwhich is a tool for converting Android's .dex format to Java's .class format. It takes the whole APK file as input and gives a JAR file which contains the .classfiles as output. After that, the manifest file (AndroidManifest.xml) is extracted from the

JAR file for further inspection. The manifest file defines, among other things, the permissions an application uses and the permissions other applications need in order to access the components of that application. Since the manifest file is in binary XML format, it is first converted back to human-readable XML using another tool called AXMLPrinter2. By looking at the manifest file, the answers to the following questions could be known:

1. Is the application asking for additional capabilities by requiring permissions?
2. What are the components that are publicly accessible by other applications?
3. Are those publicly accessible components guarded by permissions?

A list of components that have the potential of leaking capabilities is then obtained. These are the components that have extra capabilities, are publicly accessible, and are not guarded by any permissions. Finally, using inter procedural control flow graph searching and static taint checking, data paths in the source files of these components that will lead to capability leaks could be found. The source files are obtained by decompiling the class files in the JAR archive using Java Decompiler which is the latest Java decompiler.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest>
  <uses-permission />
  <permission />
  <application android:permission="...">
    <activity android:permission="..." android:exported="...">
      <intent-filter> ... </intent-filter>
    </activity>
    <service android:permission="..." android:exported="...">
      <intent-filter> ... </intent-filter>
    </service>
  </application>
</manifest>
```

Figure 1.2:General structure of AndroidManifest.xml

In the rest of this section, the two key modules, manifest file parsing and capability leak detection, are to be discussed in details.

B. Manifest File Parsing

Before going into technical details of the manifest file parsing module, the structure of the manifest file is discussed and the various tags found in it are briefly explained. After that, the high-level checking policy is introduced. Finally, the low-level details of the manifest file parsing are given.

C. The AndroidManifest.xml

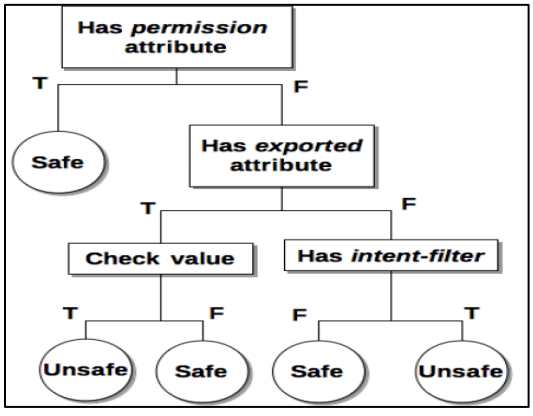
An abstract of the overall structure of the manifest file is shown above. The uses-permission tag is used to requests a permission that the application needs to be granted such that it can operate correctly. This is how an application requests for capabilities. The permission tag simply declares a security permission that can be used later to limit access to special components or features of this or other applications. The android:permission attribute of the application tag is used to declare the permission that components of other applications need to possess in order to interact with this application. This is how an application can protect itself and the capabilities it owns. Similarly, the android:permission attribute of the tag that defines a component declares the permission that components of other applications need to possess in order to interact with that component. Besides, a component can be made private by setting the android:exported attribute to false. The consequence is that such component will not be accessible by components of other applications. In case the android:exported attribute is absent, the default value of it depends on whether or not there are intent filters defined for that component (except for Content Providers, which have the default value being true). If it turns out that there is no intent filters, the value is set to false by default. Otherwise, the value is set to true. An intent filter is used to specify the types of implicit intents that a component can respond to and filter out intents that are not suitable for that component. However,developers should not rely on intent filters to protect a component because

explicit intents can always target at it no matter how the intent filters are defined.

D. The Checking Policy

The AndroidManifest.xml is parsed by the checking system to find out if:
1. The application uses at least one permission and;
2. There exists an activity or service component that is not protected by any permission and is publicly accessible
If the above two are both satisfied, components satisfying the second requirement are components that have the potential of capability leak. The focus of this research is activity and service components since, among the four kinds of components, they are the components that can be directly exploited to launch a privilege escalation attack. A component is exploitable if it can receive intents from other components and it can perform security sensitive operations given that the application it belongs to owns the needed permissions. An activity component represents a single screen with a user interface while a service component is a component that runs in the background to perform long-running operations or to perform work for remote processes. Both of them can be protected by permissions. We do not consider content providers as they cannot receive intent. We also do not consider broadcast receivers as they are just "gateways" to other components and are intended to do a very minimal amount of work.

Fig 1.3 : Decision Tree



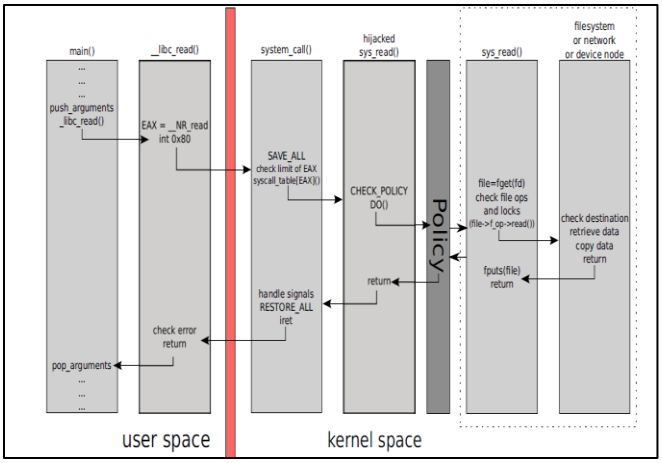


Fig 1.4: Hijacking system calls using LKM

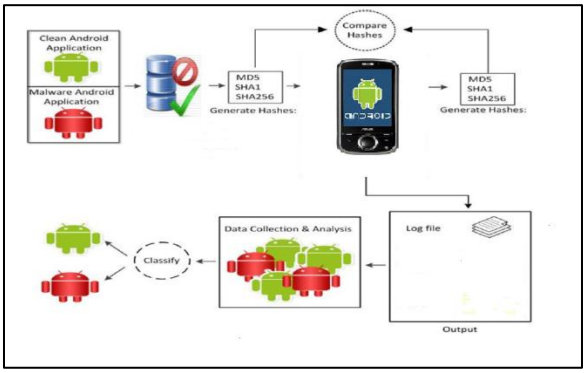


Fig 1.5 : Malware Detection System

IV.Dynamic Analysis

The LKM is intended to hijack all available system calls targeting for the ARM architecture since this is used for the Android operating system. The following pseudo code shows how the system calls are redirected:

```
asmlinkage long
new_syscall
(type1 param1, type2 param2, ...)
{
retval = orig_syscall(type1 param1, type2 param2, ...);
PRINT_TO_LOG( "syscall()", retval);
returnretval;
}
```

In the pseudo code you can see that each system call which is redirected firstly behaved like the

original system call to get the real return value. This is achieved since because we have a policy that enforces only logging of the reactions of the system call. Then the "PRINT TO LOG" prints out the gathered information to the kernel log. After this, the original return value is returned to the kernel environment so that the context of the application can proceed. In dynamic analysis the application is installed in the standard Android emulator from the Google Android SDK bundle. When the application is run for the first time, it has to be executed for a specific time. Our LKM helps us to log the behavior of the application at system-level by intercepting and logging the system calls made it. A feature vector is prepared for better analysis. The kernel module makes sure that each occurrence of a system call is logged, including those which are performed by processed other than the currently analyzed application to make sure that the complete system state is recorded and no malicious activity remains hidden.

308,com.alfasafe,1,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0
,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,1,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1
23,4,0,0,76,324,1,0

Eg. Feature Vector

We then run machine learning algorithms on the dataset obtained by collecting the feature vectors of many applications to classify the applications as malware or benign.

A. K – Nearest Neighbor Algorithm
    1. For each training example <x,f(x)>, add the example to the list of training examples.
    2. Given a query instance z to be classified :
        2.1 Let x1,x2…xk denote the k instances from training examples that are nearest to z.
        2.2 Return the class that represents the maximum of the k instances.

B. K – Means Clustering Algorithm
    1. Select k random instances from the training data subset as the centroids of the clusters C1; C2; ...Ck.
    2. For each training instance X:
        a. Compute the Euclidean distance D(Ci,X), i=1...k:Find cluster Cq that is closest to X.
        b. Assign X to Cq. Update the centroid of Cq.
    3. Repeat Step 2 until the centroids of clusters C1;C2;...Ck stabilize in terms of mean-squared- error criterion.
    4. For each test instance Z:
        a. Compute the Euclidean distance D(Ci,Z), i=1...k. Find cluster Cr that is closest to Z.
        b. Classify Z as an anomaly or a normal instance using the Threshold rule
    5. Assign Z-->1 if P(w|Z)> Threshold. Otherwise Z = 0 And Z = 1 where 0 and 1 represent normal and malware Classes.

C. J48 Decision Tree Algorithm
    1. Check for base cases
    2. For each attribute a
        Find the normalized information gain ratio from splitting on a.
    3. Let a_best be the attribute with the highest normalized information gain.
    4. Create a decision node that splits on a_best
    5. Recurse on the sublists obtained by splitting on a_best and add those nodes as children of node.

V. Practical Application

Since Android is the most widely used mobile OS and the malwares flood the application sources, user privacy and data have to be protected more severely. Our system helps the user in this respect.

VI. Summary

In this paper we have proposed the concept of automated malware detection for Android phone users to prevent any kind of data leaks, may be caused by a malicious application. The main advantage of this model is that the process will be running in background monitoring each call made by the application & authenticating it for its category. The developer‘s tool also parses the entire package file hooking on to any modules which may fall prey to privilege escalation attack or a capability leak. It gives the developers tabular and graphical results which illustrates those modules in the codes he needs to strengthen. This will also secure the sensitive information on the phone, hence motivating users to use the cell phones for monetary transactions and other personal usage.

VII. REFERENCES

[1] M. A. Bishop. The Art and Science of Computer Security. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[2] MoutazAlazab, VeelashaMoonsamy, PatrikLantzLynn. Analysis of Malicious and Benign Android Applications. In 2012 32nd International Conference on Distributed Computing Systems Workshops

[3] IkerBurguera, UrkoZurutuza, SiminNadjm-Tehrani.Crowdroid: Behavior-Based Malware Detection System
for Android.

[4] S. Forrest, S. Hofmeyr, and A. Somayaji.The evolution of system-call monitoring. In ACSAC ’08: Proceedings of the 2008 Annual Computer Security Applications Conference, pages 418–430. IEEE Computer Society, 2008.

[5] R. Love. Linux System Programming.O’Reilly, 2007.

[6] http//developer.android.com (Last cited on: 8th January 2013)

[7] http://forum.xda-developers.com (Last cited on: 2nd January 2013)

[8] https://play.google.com/store/apps (Last cited on: 7th January 2013)[9] R. Love. Linux System Programming. O’Reilly, 2007.

[9] R. Love. Linux System Programming. O’Reilly, 2007.

[10] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. IEEE Security and Privacy, 5(2):32–39, 2007.

[11] http//developer.android.com (Last cited on:25th September 2013).

[12] http://forum.xda-developers.com (Last cited on: 22th September 2013) .

[13] https://play.google.com/store/apps (Last cited on: 17th August 2013) .

[14] Wei-Meng Lee, Beginning Android 4 Application Development. Wrox, 2012.