How to Violate Android's Permission System without Violating It

Kyoung Soo Han[1)], Yeoreum Lee[1)], Biao Jiang[1)], Eul Gyu Im[2)]
Dept. of Electronics and Computer Engineering[1)],
Div. of Computer Science and Engineering[2)],
Hanyang University, Seoul, Korea
{1hanasun, lufi, dante, imeg}@hanyang.ac.kr

ABSTRACT

Android uses permissions for application security management. Android also allows inter-application communication (IAC) which enables cooperation between different applications to perform complex tasks and is a major feature that differentiates Android from its competitors. However, IAC also facilitates malicious applications to collude in an attack of privilege escalation. In this paper, we demonstrate by case studies that all IAC channels can be potentially utilized for privilege escalation attacks, and propose refinement to solve this problem by taking IAC as permissions and exposing IAC to users.

KEYWORDS

Smartphone security, Android security, Android permission, Permission-based security, Privilege escalation attack

1 INTRODUCTION

Smartphones have already become an essential part of modern life. They become increasingly powerful, and possess more and more private information of users, which makes them an attractive target for attackers.

Android[1] was announced by Google in 2007, and has become the most popular smartphone operating system since 2011[2]. Compared with Apple's iOS and Microsoft's Windows Phone, Android provides more flexibility and places less restriction on application development. Particularly, inter-application communication (IAC), empowered by inter-component communication (ICC), enables cooperation of different applications with different abilities to fulfill a task together, even without knowing each other during development. IAC facilitates complex goals to be achieved which may not even possible for operating systems without IAC, and is a major feature differentiating Android from its competitors. However, IAC also weakens application-wise isolation and introduces potential risks of privilege escalation.

We have looked into this issue and found out that every type of IAC available on Android is vulnerable for privilege escalation. In addition, we have demonstrated the vulnerability by case studies on each IAC channel and provided details of subtle points which are usually ignored. Moreover, we have also obtained further insights of the current permission system on Android and proposed the refinements.

The rest of paper is organized as follows: Section 2 provides the background knowledge of Android's framework. The attack model against Android's permission system is introduced in Section 3 with demonstrative case studies. Discussion on the finding is explained in Section 4, followed by our proposal for refinement. Section 5 summarizes related works. Section 6 concludes the paper.

2 BACKGROUNDS

2.1 Applications

Different from their ancestors, the powerfulness and flexibility for smartphones including Android are largely built on applications. An Android application is an archive file (.apk, Android package) containing its components, resources and a manifest file. The manifest file contains information about all the static components, the permission requirements and the other descriptions of the application.

An application on Android is isolated from other applications by its unique Linux user ID. It has its

own storage space and runs in its separate sandboxed process. Simultaneously, sharedUserId can make Android treat applications sharing the same user ID as the same application.

2.2 Permissions

An application requires permissions to access resources and fulfill its jobs. Android has four types of permissions: *normal*, *dangerous*, *signature*, and *signatureOrSystem*.

A *normal* permission refers to a permission that is of low risk and is automatically granted; contrarily, a dangerous permission refers to a permission that is of high risk and needs explicit approval from users. A *signature* permission is only granted to applications signed with the same certificate as those in the system image, and a *signatureOrSystem* permission is only granted to applications that are signed by the same certificate as those in the system image or are in the system image.

Among the above, only *dangerous* permissions are exposed to and decided by users. A user accepts all permissions proposed by an application in order to install it, and the system enforces the permissions during the runtime. In this way, a user is informed about the permissions an application requires.

2.3 Components

An application's functionalities come from its components. There are four types of components in Android: *Activity*, *Service*, *BroadcastReceiver* and *ContentProvider*.

An *Activity* component interacts with users and is the only kind of components which runs in the foreground. A *Service* component performs supportive operations, and a *BroadcastReceiver* component responds to broadcasting events, both in the background. A *ContentProvider* component offers programmatic interfaces for data sharing, possibly across applications.

Any component may or may not cast permission requirements when it is accessed across applications, and may even completely deny all external requests from other applications by setting its exported flag to false. It is worth noting that extra security settings for components are decided by developers in order to protect the components and the applications they are a part of.

2.4 Inter-Component Communication

On one hand, Activity components, Service components and BroadcastReceiver components (i.e. except for ContentProvider components) use *Intents* for inter-component communication.

Intents are either explicit or implicit. The former means that the initiator of the Intent specifies the responder, whereas the latter means that no specific responder is designated but any qualified component is allowed to complete the job.

If Activity components or Service components are targeted, only one component will be the final responder; if BroadcastReceiver components are targeted, all applicable ones will respond (if not aborted by one in the middle).

On the other hand, ContentProvider components open an interface to share data, possibly across applications, and other components fetch data through the interface.

Inter-component communication empowers inter-application communication.

2.5 Tasks

In contrast to the intuition that the runtime user experience is organized per application, Android actually manages it by *task*, which is a collection of Activity components that root from the same starting point and are launched in a chain. A task is internally implemented as a *back stack* which arranges the root Activity component in the bottom and the current Activity component on the top. Whenever a user starts an application, what is actually started is the "main" Activity component, and a task associated with it is maintained by the system. Later, if the user presses the home key, the task loses focus and only returns to the screen again when the same application starting the task is launched again. An easy way to return to recent tasks is from the application history which can be activated by long pressing the home key, where only "root" applications which have started tasks are recorded.

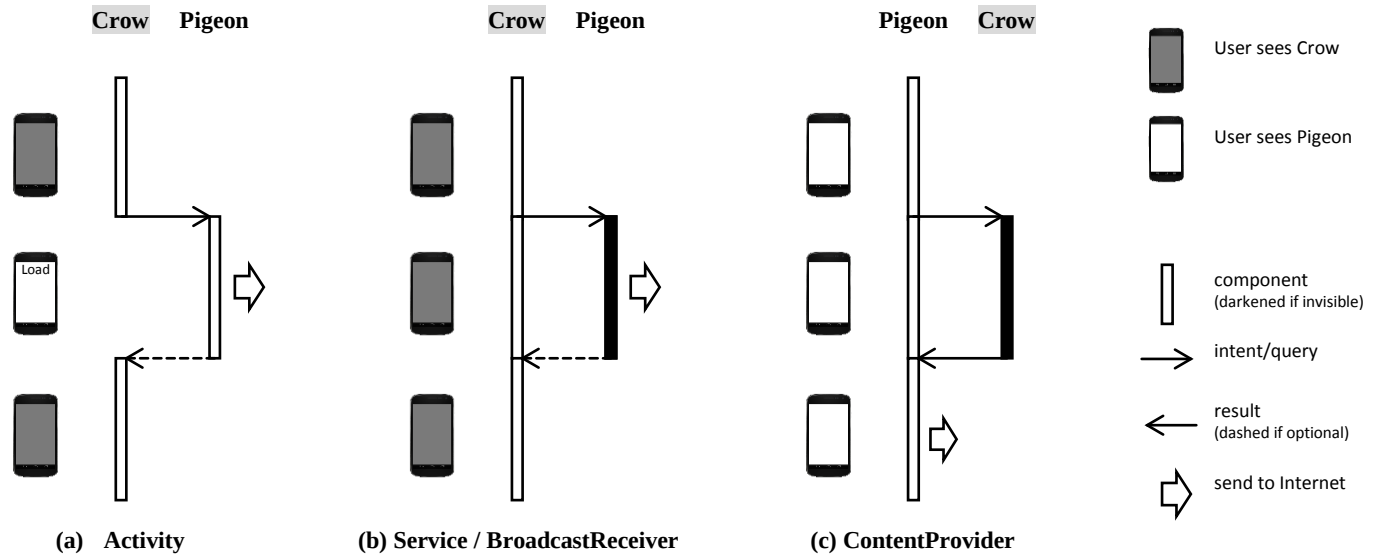

Figure 1. Case Study Illustration

3 ATTACK MODEL

While IAC enables components from different applications of different abilities and permissions to cooperate together, it also facilitates malicious developers to work around the permission controls of Android.

The task model of Android's runtime management has already showed some clues by allowing combining Activity components from different applications as if they were a single application to achieve a complex goal. Malicious applications with different permissions can also conclude to utilize this combinational powerfulness to achieve a privilege escalation attack.

We will demonstrate how this powerful feature can be abused to bypass the security enforcements of Android's permission system by case studies. Four cases are included in our case studies, and attacks through each type of components are demonstrated respectively.

We have implemented four pairs of applications, each pair of which includes two applications, called Crow and Pigeon. The full package name of Crow is *corvidae.crow*, and it requires a single dangerous permission to access private information, "Read Contact Data (`android.permission.READ_CONTACTS`)". It represents a privacy-sensitive application. Because the request to read contact data is the only permission required, it is still generally regarded as a safe application. The full package name of Pigeon is *columbidae.pigeon*, and it also requires a single dangerous permission, "Full Internet Access (`android.permission.INTERNET`)". It represents an ordinary application with network access, for example, an ad-supported free application. Both of them appear to be harmless and even obey the least privilege principle. However, if both of them are installed on the same device, they will collude for privilege escalation and be able to leak contact data into the Internet.

3.1 Case 1: Activity

As shown in Figure 1(a), when running, Crow reads contact data and sends an Intent containing the contact to one of Pigeon's Activity components which called Deliver explicitly. Upon receiving the Intent, Deliver becomes the foreground component, and at the same time, Pigeon extracts the data from the Intent, sends it to a server in the Internet, and finishes itself. The original Activity component which initiates the Intent then comes back to the foreground again.

In this case, the foreground is changed. However, as long as Deliver disguises itself as an intermediate "loading" or "processing" Activity component, the truth is of little chance to be discovered by users. In addition, the application history does not record intermediate Activity components and no trace is left over. Even if Pigeon is started separately by the

user, it will live in another task in the bottom of a separate stack, and nothing will appear to be suspicious.
It is worth noting that explicit Intents are used to avoid prompting users for selection and thus to avoid being noticed. If there are multiple Deliver-alike Activity components, one-by-one polling can be used.

3.2 Case 2: Service

As shown in Figure 2(b), Crow starts a Service component of Pigeon explicitly by an Intent containing contact data and keeps running. As soon as Pigeon receives the Intent, it begins running parallel to Crow and uploading any received contact data, and upon completion, it finishes itself quietly.
In this case, because the uploading happens in the background and no user interface is changed, it is obviously difficult to be noticed by users.

3.3 Case 3: BroadcastReceiver

The procedure is almost identical to Case 2, and Figure 1(b) is reused for illustration. The only difference is the use of a BroadcastReceiver component instead of a Service component.
In this case, implicit Intents are also usable, as broadcasting is targeted at every receiver. In case of data upload, it is not necessary to submit the same content multiple times, and as a result, the first receiver should abort the broadcast from propagating to the other receivers.
In addition, it is possible to register a BroadcastReceiver component dynamically at runtime. This allows malicious developers to be able to eliminate the details of BroadcastReceiver components in the manifest file. Although this makes no difference for our demonstration, it may completely circumvent any security systems which depend on manifest files.

3.4 Case 4: ContentProvider

As shown in Figure 1(c), Crow defines an interface for querying contact data obtained, and opens the interface to Pigeon. When Pigeon is running, it communicates with Crow through the interface to obtain contact data and sends the results of its query to the Internet.
In this case, instead of using Intents for communication, Crow directly opens an interface for data provision and Pigeon fetches data from it. Again, no anomaly appears to users' screens during the whole procedure.
It is worth noting that though in case studies two applications exercise their permissions within a short time frame, it is not necessary; for example, in the first case for Activity, Pigeon can just save all the contact data it receives and transmits the data later. In addition, more complex attacks involving more than two applications are also possible. Furthermore, the targets of attacks are not limited to system resources, for instance, a public component protected by permissions.

4 DISCUSSIONS

Case studies in the previous section have proved that all four types of components can be exploited to provide extra permissions and the effects to unauthorized applications. The violation of permission borders is achieved by implementing a component of an application as a proxy to provide resources to other applications which do not have the permissions to access those resources.
This security hole is caused by the fact that the permission requirement of custom components is decided by developers; if developers want to abuse the permissions granted to their applications intentionally, the system has no means to prevent the abuse.

4.1 Observation

By investigating about this security hole further, we have obtained more insights.

Privilege escalation is a feature for IAC: Android allows applications to communication with each other. IAC, by cooperation of applications with different abilities, enables a lot of use cases that are not applicable in smartphone operating systems without IAC, and it is a major feature that differentiates Android from its competitors. However, different abilities usually require different permissions, and the cooperation of applications with different permissions results in privilege

escalation. Thus, privilege escalation should be regarded as a feature of Android.

IAC should be treated in the same way as permissions: Privilege escalation is a natural result of IAC. The current security risk of privilege escalation Android faces is resulted from the fact that Android hides all the details of IAC from users. Straightforwardly, such a risk can be solved by exposing IAC and taking IAC as a permission. More specifically, any application that requests the ability to start IAC should not only ask for the permissions it requires (as it does now) but also request any privilege that might be involved in IAC. Although such practice obviously increases the number of permissions an application requires on average, it just reveals the true picture to users. Think it in a different way: the number of permissions an application needs under such practice is exactly the same number it will require if there is no IAC and it is still implemented to achieve the same goal. To avoid the problem of over privilege as more permissions are granted than the application itself needs, permissions for IAC should be declared in a different way so that the application itself cannot use them. In this way, the principle of least privilege is guaranteed.

Users' choices rule: It is impossible to cover all kinds of use cases. This has already been proved by inevitable false positives from various researches on Android's permission system. Users have to get involved in the decision process about whether certain IAC should be allowed. Because no pre-defined policies are possible to cover all use cases, users should be in the position of final decision.

Components conduct communication: IAC is fulfilled by components of applications, rather than applications themselves. A single component usually takes less permission than the whole application, especially for public interfaces for IAC, which are more likely to be designed and implemented carefully. Component level granularity will greatly help to reduce the number of permissions involved in IAC.

IAC interfaces are determined at development time: The decision about which components are exported is made during development. Of course, the actual pairs of communicating components are possible for late binding at runtime, for example, by requesting a generic action type, such as sharing. On a side note, it should be forbidden to dynamically generate any public interface. In order to achieve the dynamic property of an interface, a toggle for public interfaces can be introduced. In this way, all public interfaces of an application are statically and clearly visible in the manifest file, which contains the description of the application.

4.2 Refinement

According to the observation above, it might be a high time for refinement for an IAC enabled permission system for Android. The proposed refinement defines three levels of permissions within an application: application level, inter-application level, and component level.

1) Application Level

This level's permissions are a global limit for the application and its components. This level is very similar to the current permission system of Android, except for a few advanced features:

First, different priorities for permissions: a developer can assign different priorities to the permissions of an application. For a permission with the highest priority which is regarded by the developer as essential for the application, a user has to grant the permission to the application in order to install and use it. For example, a free application supported by ads will request network access as its essential permission. For an optional permission, a user can choose to permanently allow or deny it or decide per request. The configuration for all the optional permissions can be changed any time. To improve user experience, policy-based assistance can be incorporated, so that dangerous permissions are set to decide per request whereas others are granted by default.

Second, least privilege: an application's permissions should be recalculated upon submission to Google Play Store for Android as the current practice of Windows Phone Marketplace[3], and it

is recommended to honor the decision of a developer who intentionally leaves out a permission for an application. This is based on the assumption that the official algorithm knows the permissions better while the developer knows the application better. If the algorithm detects an unused permission and removes it, this certainly will not impact the application to any extent; if the developer leaves out a permission intentionally, the developer is supposed to have tested the application without the specific permission. An offline calculation tool should be provided through the software development kit.

The best practice for a developer to determine an application's permission is to run the offline calculation first and then mark essential permissions and remove unnecessary ones which probably introduced by unused parts of 3rd-party libraries.

For example, a camera application may regard the main camera as essential and at the same time require the front camera optionally. Code snippet for declaration of application level permissions of such a camera application is shown in Figure 2.

```
<uses-permission
  name="android.permission.CAMERA"
  optional="false" />
<uses-permission
  name="android.permission.FRONT_CAMERA"
  optional="true" />
```

Figure 2. Code Snippet for Application Level Permissions

2) Inter-Application Level

Inter-application level plays the core role in the refinement. Instead of hiding IAC details from users, the proposed refinement requires an application to state all out-going communication where the application acts as a requester and all permissions the requested component should have on a basis of per communication type. It is worth noting that the permissions claimed in inter-application level are only used by requested components in other applications but not used by the application itself, so that these permissions will not extend the permissions for the application itself.

When an IAC request is issued, the system will not only match the basic criteria such as the name or the general category of the requested component, but also check the permissions of the matched component which is supposed to request fewer permissions than or the same permissions as what the request states. Only a component that meets both the criteria and the permissions will be able to conduct the IAC.

All the inter-application level permissions are deniable and can be closed on per communication basis. If all IAC is forbidden, the applications are completely isolated from each other.

To make developers' life a little bit easier, any IAC to system applications does not need to be specially claimed. As long as a system application is not faultily implemented, it is unlikely to collude in an attack against the permission system.

The best practice for a developer is to set a system application as a Plan B whenever possible, thus even if the specific inter-application level permissions are denied or no matching component is found, the IAC will still be established.

For example, the camera application mentioned above may want to let users share photos easily, and it can be simply implemented by an inter-application sharing request. The IAC request may need internet access and the code snippet is shown in Figure 3.

```
<uses-inter-application-communication>
  <action
    name="android.action.SHARE" />
  <uses-permission
    name="android.permission.INTERNET" />
</uses-inter-application-communication>
```

Figure 3. Code Snippet for Inter-Application Level Permissions

3) Component Level

This level's permissions put limitation on components for IAC. Similar to the permissions for an application, the permissions for a component are determined by a hybrid approach where automatic calculation gives the range of permissions and developers modify the result. Priorities for permissions are also used in component level. By making nonessential permissions optional, a component is more likely to meet the permission requirement of inter-application request. Moreover,

optional permissions on component level are helpful for developers to reuse components; instead of re-implement a similar component with fewer permissions for better public exposure, a develop only needs to check the availability of permissions and make sure a component is working in a downgraded but correct way when some permissions are not available.

Normally, component level permissions are not exposed to ordinary users. But they can be easily found by developers who want to set up communication to the application.

For example, a social networking service (SNS) application usually provides an interface for sharing. It may require internet access and optional GPS access, the latter of which is used for check-in. Figure 4 shows code snippet for such an exported interface as Activity.

```
<activity exported=”true” >
  <intent-filter
    name=”android.action.SHARE”/>
  <uses-permission
    name=”android.permission.INTERNET”
    optional=”false” />
  <uses-permission
    name=”android.permission.FINE_LOCATION”
    optional=”true” />
</activity>
```

Figure 4. Code Snippet for Component Level Permissions

5 RELATED WORK

The official Android Developers website[1] is the ultimate source for application development guides and references.

A specialized example of privilege escalation attacks against components protected by permissions is given in [4], but it does not cover general cases. XManDroid[5] and IPC Inspection for Android[6] aim to prevent applications from extending their permissions and forming dangerous combinations by IAC. TaintDroid[7] is a multi-level information-flow tracking system to prevent sensitive information propagating across applications. AppFence[8] also provides tainted analysis and blocks network transmission of private data. Kirin[9] is a policy-based system to avoid applications with dangerous permissions. SAINT[10], an install-time and runtime hybrid system, also applies permission check upon installation. However, it is impossible to cover all use cases by policies and the systems mentioned above are prone to false positives; besides, it is difficult to achieve the balance of security and effectiveness, because strict policies largely limit Android’s differentiating feature.

ComDroid[11] is an analysis tool for IAC vulnerabilities based on Dalvik executables. Quire[12] provides call chain verification to defend against confused deputy attacks but cannot stop intentional collusion. PScout[13] shows a relationship data mapping information between API call and permission. Apex[14] and MockDroid[15] both enables selective permissions by either revoking permissions entirely or providing dummy data when applicable.

6 CONCLUSION

Android’s inter-application communication enables complex tasks to be done by cooperation of different applications with different abilities, and is a major feature that differentiates Android from its competitors. However, IAC also utilizes malicious applications to collude in an attack against the privilege system. Our case studies have proved that all types of IAC available for Android are vulnerable for privilege escalation attacks. The risk of privilege escalation is mainly resulted from the fact that Android hides all IAC details from users. To enhance the security of Android, refinement for Android’s permission system is proposed; particularly, inter-application level permissions are introduces, which plays a core role to prevent privilege escalation.

7 ACKNOWLEDGMENTS

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MEST) (No. 20120006492).

8 REFERENCES

1. Android Developers. http://developer.android.com.
2. How Google And Apple Won The Smartphone Wars, http://techcrunch.com/2012/01/02/chart-google-apple-smartphone-wars/
3. MSDN, How to: Determine Application Capabilities. http://msdn.microsoft.com/en-us/library/gg180730(v=vs.92).aspx
4. Davi, L., Dmitrienko A., Sadeghi, A.R., Winandy, M.: Privilege Escalation Attacks on Android. In: Burmester M. et al. (eds.) ISC 2010. LNCS, vol. 6531, pp. 346-360, Springer, Heidelberg (2011).
5. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.R.: XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks. Technical Report, Technische Universität Darmstadt (2011).
6. Felt, A.P., Wang, H.J., Moshchuk, A., Hanna, S., Chin, E.: Permission Re-Delegation: Attacks and Defenses. In: Proc. 20th USENIX Security Symposium (USENIX Security ’11), San Francisco (2011).
7. 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: Proc. 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’10), Vancouver (2010).
8. Hornyack, P., Han, S., Jung, J., Schechter, S., Wetherall, D.: These Aren’t the Droids You’re Looking For: Retrofitting Android to Protect Data from Imperious Applications. In: Proc. 18th ACM Conference on Computer and Communications Security (CCS ’11), pp. 639-652, New York (2011).
9. Enck, W., Ongtang, M., McDaniel, P.: On Lightweight Mobile Phone Application Certification. In: Proc. 16th ACM Conference on Computer and Communications Security (CCS ’09), pp. 235-245, Chicago (2009).
10. Ongtang, M., McLaughlin, S., Enck, W., McDaniel, P.: Semantically Rich Application-Centric Security in Android. In: Proc. 25th Annual Computer Security Applications Conference (ACSAC ’09), pp. 340-349, Honolulu (2009).
11. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing Inter-Application Communication in Android. In: Proc. 9th International Conference on Mobile Systems, Applications, and Services, pp. 239-252, Bethesda (2011).
12. Dietz, M., Shekhar, S., Pisetsky, Y., Shu, A., Wallach, D.S.: Quire: Lightweight Provenance for Smart Phone Operating Systems. In: Proc. 20th USENIX Security Symposium (USENIX Security ’11), San Francisco (2011).
13. Au, K.W.Y., Zhou, Y.F., Huang, Z., Lie, D.: PScout: Analyzing the Android Permission Specification. In: Proc. 19th ACM Conference on Computer and Communications Security (CCS ’12), pp. 217-228, Raleigh (2012).
14. Nauman, M., Khan, S., Zhang, X.: Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints. In: Proc. 5th ACM Symposium on Information, Computer and Communications Security (ASIACCS ‘10), pp. 328-332, Beijing (2010).
15. Beresford, A.R., Rice, A., Skehin, N., Sohan, R.: MockDroid: Trading Privacy for Application Functionality on Smartphones. In: Proc. 12th Workshop on Mobile Computing Systems and Applications (HotMobile ’11), pp. 49-54, Phoenix (2011).