ATLANTIS PRESS
Advances in Engineering Research, volume 118
2nd International Conference on Automation, Mechanical Control and Computational Engineering (AMCCE 2017)

An Enhancement Method for Android Permission Mechanism based on Context

Guomin Hou, Yi Zhuang* and Jiaye Pan
College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China
zy16@nuaa.edu.cn

Keywords: Android, permission mechanism, access control, privacy protection.

Abstract: A context-related permission model is proposed for the few control over the permission use of the Android permission mechanism. Also extend the Android system to implement a context-related permission executing system based on the context-related permission model. By enhancing the Android permission mechanism , the context-related permission executing system could identify the real origin of permission request and have a fine-grained control over permission use, which can effectively prevent the intra-application malicious code attack. Experimental results show that the context-related model can effectively control the permission use of application and introduce little performance overhead compared to Android system.

1. Introduction

With the rapid development of the mobile Internet, Android smart phones have gradually become an indispensable part of people's life. According to data report released by the IDC, as of the second quarter in 2016, Android operating system had 87.6% share of the smart phone market [1], which occupy the absolute dominant position. While the popularity of smart phone increases rapidly, its security problems are increasingly serious: the frequent leakage of user privacy and malicious applications one after another.

Prior to the Android 6.0 release, Android system provides a permission-based access control mechanism[2]. However, due to the coarse-grained defect of the Android permission mechanism, when install the application, users have to agree all the permission requests otherwise they will be unable to install the application. After the application has been installed, users will have no idea about how and when the installed applications use the granted permissions. Besides, because the existing permission mechanism can’t identify the real source of a permission request, Android system thus can’t effectively prevent such intra-app malicious code attack. Although the latest version of Android realize the function of instant authorization of permission use, it also has defect: once granted, the permission will be permanently used by application unless the user reset it. For some high-risk permissions, the system still can’t authorize dynamically when the application is ready to use the permissions, which will bring risks to the user privacy. What’s more, because Android system has many versions and the problem of fragmentation is serious, many users still fail to update their devices to the latest version.

For the defects of Android permission mechanism, there are a lot of research. By extending the Android permission model, Apex [3] provides a policy enforcement framework that allows users to selectively grant permission when the application is installed. Furthermore, Apex enables users to add constraints to system resource use. GrantDroid [4] provides a way to authorize Android permissions instantly by intercepting all permission use requests and detecting malicious permission use by using a feature database based on a full set of valid malware permissions usage, and ultimately achieves the function of instant authorization of permission. Christian Jung et al. proposed a context-sensitive policy enforcement framework [5], which takes the different requirements for device security in different usage scenarios (e.g., at home, in the company, or during travelling) into account. However, in the existing research work, there is little work in the realization of the

fine-grained user-centric control over the permission use and the prevention of intra-app malicious code attack.

Aiming at the above problem of Android permission mechanism, this paper proposes a context-related permission execution model and designs a context-related permission execution system named CReDroid based on this model. By context-related permission execution model, CReDroid could have a fine-grained user-centric control over the permission use. Besides, the context-related permission execution model could identify the real source of permission use request, which will be effective to prevent application malicious code attack. In the end, we evaluate the CReDroid system from two aspects of effectiveness and performance overhead and the experimental results show that CReDroid can have fine-grained control over permission use. Besides, compared to the Android system, CReDroid system introduces little performance overhead as well.

2. A Context-related Permission Model

2.1 Threat Analysis

In order to launch an attack and hide their behavior of abusing permissions, the attacker will often choose the applications which include target permissions and then, inject malicious code into them [6]. In general, the steps of Android application malicious code injection are shown in Figure 1. When the re-packaged applications are installed, the malicious code will run as a benign application and "legitimately" use the target permissions to launch the attack. However, due to the coarse-grained defects of the permission mechanism, Android system will appear powerless in response to this attack.

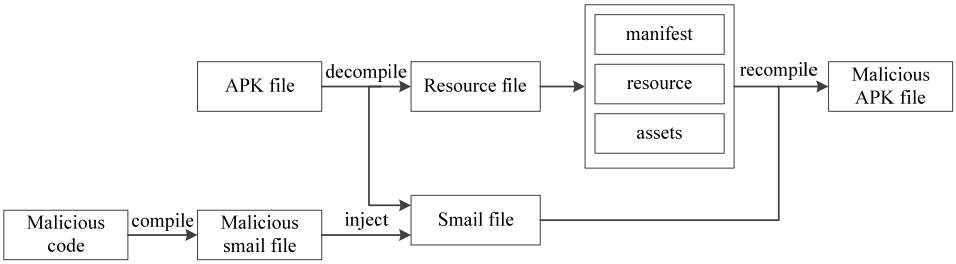

Fig. 1 The steps of application malicious code injection

Based on the risk Android permission mechanism is faced with, this paper considers the following attack scenario: an attacker can hide his identity and meanwhile, launch an effective malicious attack, where an effective malicious attack means that the Android permission mechanism does not recognize the permission request is from the third-party malicious code. In addition, this paper assumes that the application are benign ones before injected. How to enhance the permission mechanism and making it to overcome the above threat is our main work in this paper.

2.2 A Context-related Permission Model

As one of the main security mechanism in Android system, the permission mechanism mainly includes two modules: ActivityManagerService and PackageManagerService. Among them, ActivityManagerService module is mainly responsible for the identity check of the application. When the application succeeds the identity check, the PackageManagerService module will look up the permission table according to the Uid of the requester, and judges whether the application has the corresponding permission. If so, Android system will approve of the request of permission use. Otherwise, the request will be denied. Based on the principle of Android permissions mechanism, this paper abstracts the permission mechanism as the following permission check model.

Definition 1. The existing Android permission mechanism can be abstracted as a triplet.

$$PCM = (Uid, Permission, Action)$$

Where:

- *Uid* is the identity of the application. When install the application, the system will randomly assign an identity (Uid, user id) to the application and different applications would have different identities. Once the application is installed, its initial assigned identity will remain the same until the application is uninstalled.

- *Permission* is the permission to check. After the application is installed, the PackageManagerService module builds a permission table for each application. The permission to check is one of the permissions in the permission table.
- *Action* is the decision made by the system for the permission usage request. $Action = \{Allow, Deny\}$, and if the permission check is approved, the *Action* = *Allow*; else, *Action* = *Deny*.

According to *PCM*, it is only related to the identity and the permission table of the application whether to approve of the permission use request or not. Thus, the system will inevitably suffer the attack. In order to effectively prevent the malicious code attack, this paper introduces the role of context and abstracts a series of function calls that triggers the permission request as the application context information. Making use of the context information to enhance the permission mechanism and construct a context-related permission check model, which presents as follows.

Definition 2. The existing permission checking model is extended to a context-related permission check model by introduce the role of context, which can be abstracted as the following five-tuple model.

$$CRPCM = (Uid, Context, \delta, Permission, Action)$$

Where:

- *Uid, Permission.* Their definition are same with Definition 1.
- *Context* is a series of function calls that trigger the permission request. $Context = \{f_1, f_2, ..., f_i, ..., f_n\}$ and f_i denotes one of the functions in the series, $i = 1, 2, ..., n$.
- δ is function that is used to calculate the tag value of the function call series and $\delta : con \in Context \rightarrow pcc$,where *con* is a function call series and *pcc* is its corresponding tag value.
- *Action* is the decision made by the system for the permission usage request. $Action = \{Allow, Deny, Notify\}$. Unlike *PCM*, the *CRPCM* adds an authorization decision of "notify" to each permission usage request. For those permission policies with "notify" option, the users will be informed to authorize dynamically when the corresponding permission use request is triggered.

After the permission mechanism is enhanced by the context, the system will check not only the application identity and the permissions table. The context information of the requested permission will also be checked.

3. Design of CReDroid

In order to solve the coarse-grained defects in the Android permission mechanism, this paper designs a context-related permission execution system named CReDroid based on the *CRPCM*.

3.1 Design of System Architecture

In order to implement a context-related permission execution system, this article extends Android architecture and modify the application layer and application framework layer of Android. The overall architecture of the designed context-related permission execution system CReDroid is shown in Figure 2.

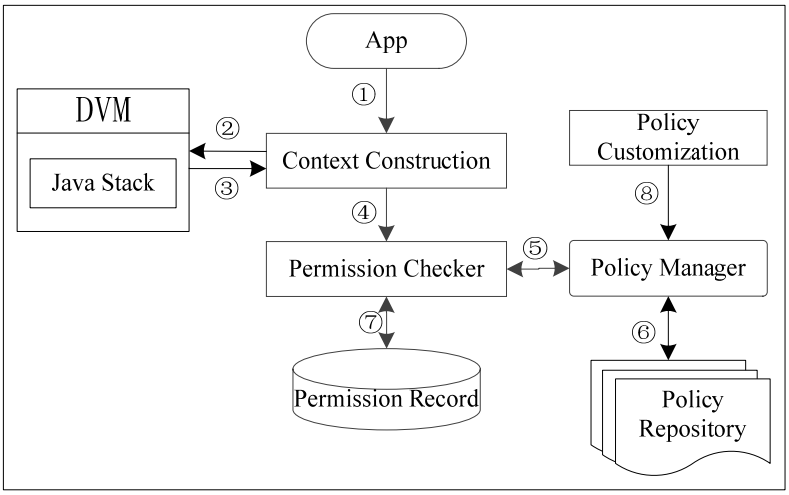

Fig. 2 The architecture of CReDroid

The system has three core modules: (1) Context Construction, which is responsible for constructing the context of each permission usage request; (2) Permission Checker, which realizes the function of context-related permission check; (3) Policy Manager, which realizes the function of adding, deleting, modifying, querying permission policy in the policy repository.

1) Context Construction

The Android system provides a Dalvik Virtual Machine (DVM) running environment for each application instance. When the application is running in DVM and triggers a permission use request, the Context Construction module needs to perform the context-construction operation. The context construction module will first query the function call stack located in the DVM and extract the function call series which triggers the permission use request. Then, according to the function call series, constructs the context information of the requested permission.

2) Permission Checker

Permission Checker is mainly responsible for the permission check. When received the context information from the Context Construction, Permission Checker will pass the Uid of the application, the requested permission and the corresponding context information to the Policy Manager. Besides, Permission Checker will also do permission check in the Permission Record just like Android permission mechanism do.

3) Policy Manager

As one of the main modules of CReDroid, Policy Manager provides guarantee for the implementation of context-related permission check. When develop the application, the developer or system administrator had better to provide permission use policies for their some high-risk permissions and when the application is being installed, the Policy Manager will read permission policy file from apk package and write it into Policy Repository. Besides, users can also formulate permission policy to control permission use based on their own needs.

The above three modules constitute the main structure of the system. When an application is running, its permission request will be redirected to the Context Construction (step 1); The Context Construction will query in the Java function calling stack(step 2) and extracts the series of function calls that trigger the permission request(step 3);According to the function series, the Context Construction builds the application context and passes it to the permission checker (step 4); Then, the Permission Checker passes the Uid, the requested permission and its corresponding application context to the Policy Manager (step 5) and performs policy matching in the Policy Repository (step 6); If the match succeeds, the operation will be performed according to the policy; otherwise, the permission check will be reverted to the native permission mechanism (step 7). Besides, users can make use of the policy customization software to dynamically specify application permissions policies and write them to the Policy Repository through the Policy Manager (step 8).

3.2 Design of context-related permission policy

In order to implement the framework of the context-related permission policy, we need to design a policy rule to facilitate the users to formulate different permission policies according to their own needs. From the perspective of user usability and system performance, the permission policy should not be too complex. Otherwise, it will affect the user experience and result in system performance degradation, not feasible. Therefore, this paper designs an easy-to-use application permission policy rule and describes the rule using the Backus-Naur Form [7] as follows.

$$< policy >::=< uid >< permission >< context >< action >$$

$$< action >::= allow \mid deny \mid notify$$

$$< contex >::= pcc \mid < func - sequence >$$

$$< func - sequence >::=< func\_1 >< func\_2 > ...$$

According to the *CRPCM* model, each application permission policy consists of four parts: an application identifier *Uid*, a permission, the corresponding action and context information. The CReDroid system mainly supports three permission actions: allow, deny, and notify. For the permission policy with allow or deny option, the system will respond to the permission use requests strictly in accordance with the policy. For those permission policies with notify option, the request will be authorized dynamically by users. For each application permission policy, the context part can be omitted. If an application has two or more policies for the same permission, the policies with context part has priority over the ones without context part. Typically, the permission policies provided by the software developer or system administrator will contain context information, which will be helpful to protect the high-risk permissions of the application and could avoid malicious code attack. The application permission policies formulated by the user will not have the context part and the user could have a fine-grained control over permission use.

According to the above permission policy rules, this paper uses XML language as the definition language of policy. As an extensible markup language, XML is more suitable for dealing with documents with structured information because of its simplicity and extensibility. An example of permission policy is shown in Figure 3.

```
<?xml version="1.0" encoding="utf-8"?>
<Policies>
        <Policy uid="test01" permission="android.permission.INTERNET" pcc="*">
                <action>allow</action>
        </Policy>
        <Policy uid="test01" permission="android.permission.READ_CONTACTS" pcc="*">
                <action>deny</action>
        </Policy>
        <Policy uid="test01" permission="android.permission.CALL_PHONE" pcc="*">
                <action>notify</action>
        </Policy>
</Policies>
```

Fig. 3 An example file of permission policy

3.3 Construction of Context

In order to identify the real source of the permission request effectively, CReDroid system abstracts the function call series that trigger the permission request to the application context information. However, considering that some function call series are too long, if they are directly abstracted as application context, it is not only too complex to construct the permission policy, but also cost a lot of time for policy match. Not feasible. Therefore, there is an urgent need for a tag value to replace the complex function call series to improve the feasibility of the context-related permission check.

This paper uses a method called Probabilistic Calling Context (PCC) [8] to calculate the tag value for each function call series, and replaces the function call series with this value. When triggering a permission usage request, the context construction extracts the series of function calls by traversing the function call stack in the dalvik virtual machine, and using formula (1) to calculate the tag value:

$$pcc = pcc' * 3 + cs \tag{1}$$

Where, pcc' is the *pcc* value of the series of function calls in the function call stack which remove the top function from stack, and *cs* is the offset of the current function relative to the beginning of the dex file. CReDroid will take advantage of this recursive method to calculate the value of the current function call series. Related researches show that the above method can calculate a unique *pcc* value for each function call sequence that triggers a permission usage request, and ensures that different sequences of function calls have different *pcc* values.

4. Experiment and performance evaluation

4.1 Experimental Environment

In order to realize the context-related permission check model, this paper extended the application layer and the application framework layer of Android5.1 system. Compile the modified source code and use the Android emulator as the experimental platform to validate the effectiveness and performance overhead of the context-related permission check model.

The hardware environment of the experiment is as follows: the operating system is Ubuntu 15.04; CPU is Intel (R) Core (TM) i5-3470 CPU@3.20GHZ; the memory is 6.00GB. We used Android Studio integrated development tool to modify Android source code.

4.2 Effectiveness

In order to test the effectiveness of the context-related permission mechanism, we select malware softwares in Drebin malware dataset [9] as the test object.

In the Drebin malware dataset, three malware families with large sample size are selected, as shown in the first column of Table 1. By means of the malicious code detection tool DroidBox [10], we divide malware into different groups according to their malicious behaviors, as shown in the second column of Table 1. According to the malicious behaviors of malware, we derive their required permission, as shown in the third column of Table 1. According to the permissions required by malware applications, formulate permission policy for every required permission and choose notify option for these policies so that when malicious applications trigger permission usage request, whether CReDroid system pops up a dialog asking the user to authorize dynamically or not. Meanwhile, runs TaintDroid[11] on CReDroid system to monitor whether there is tainted data that leaves the system. In the experiment, according to the number of samples corresponding to the malware family, the number of samples to be tested is shown in the fourth column in Table 1. Successful interception of a sample software malicious acts recorded as the number of TP (True Positive), otherwise recorded as FN (False Negative), as shown respectively in the fifth and sixth columns of Table 1. Finally, the interception rate of CReDroid in the face of malware malicious behavior is obtained according to TP and sample number, as shown in the seventh column of Table 1. The test results are shown in Table 1.

Table 1 The effectiveness test of CRPCM

Malware Family	Malicious Behavior	Permission	Samples	TP	FN	Rate
DroidKungFu	IMEI	READ_PHONE_STATE	100	92	8	92%
	PHONE_NUM	READ_PHONE_STATE	80	75	5	94%
	IMSI	READ_PHONE_STATE	100	86	14	86%
MobileTx	SEND_SMS	SEND_SMS	100	97	3	97%
	IMEI	READ_PHONE_STATE	90	81	9	90%
Adrd	PHONE_NUM	READ_PHONE_STATE	100	89	11	89%

The results show that CReDroid has high interception rate for malicious permission use request of malware from different malware families. That is to say the *CRPCM* can effectively control permission use in a fine-grained manner.

From the Table 1, the worst interception rate is 86% occurred in the DroidKungFu family. For the problem that CReDroid doesn't pop up a dialog when the malware use the restricted permissions, we choose a malware named FaceScanner(the value of MD5 is dd949b4bcafff324a2aee9c6080368ba) as the research object. Making use of ApkTool [12] and analyzing the smali code, we find that during running ,FaceScanner will communicate with another malware to get the IMEI of the device and itself doesn't steal IMEI directly. If uninstall the communicated application, the FaceScanner will behave normly. Thus, the system doesn't pop up the dialog. Do reverse analysis of the rest ones which are not intercepted, we find similar reasons. For this kind of malicious permission usage behavior between applications, the CRCPM proposed in this paper can't effectively solve it and further research is needed in the later period.

4.3 Performance Overhead

Compared with the Android permission mechanism, the introduced performance overhead of CRPCM mainly comes from the Context Construction module and the Policy Manager module.

In order to assess the additional performance overhead introduced by the CRPCM, we conducted a performance test of the Android system and the CReDroid system using six common performance test tools. The experimental results are shown in Figure 4. Experimental results show that, under the test of BenchmarkPi tool, the additional performance overhead of CReDroid is most obvious. However, even in the worst case, CReDroid has less than 3% additional performance overhead, which is in the acceptable range.

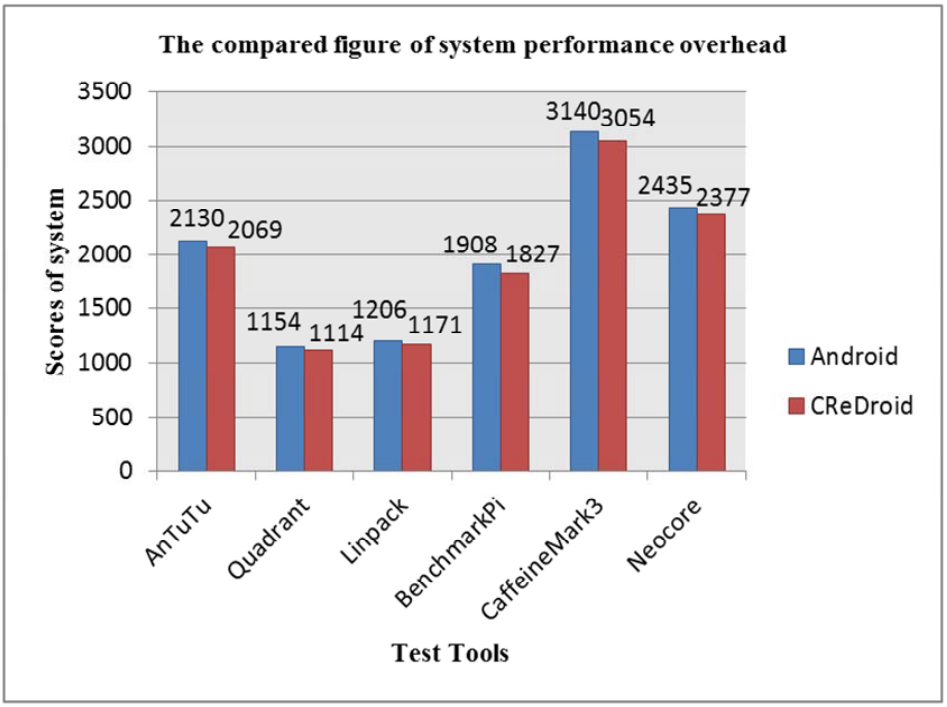

Fig. 4 The compared performance overhead of two systems

In order to test the time cost introduced by the context construction and policy lookup, we develop a test application named Test01 and Test01 will get the IMEI value of the device. In order to avoid the time error introduced artificially, select "Allow" option for the action of the permission policy. When develop the Test01, we make it loop 100 times to obtain the IMEI and every time costed will be recorded in a file. Run the Test01 on both Android and CReDroid, and the average time of 100 times access is 0.74ms for Android, 1.83ms for CReDroid. Thus, the additional performance overhead is 1.09ms. Considering the fact that the application does not need to use permissions frequently, the additional performance is within the acceptable range.

The experimental results show that, compared to the native Android system, whether in terms of overall performance overhead or the additional performance due to context construction and policy lookup, the additional performance overhead of CReDroid system are both in acceptable range.

5. Conclusions

In this paper, we design a context-related permission enforcement model for the problem that the Android permission mechanism can't effectively control the resource usage behavior of applications. Based on this model, a context-related permission executing system CReDroid is designed and implemented. Compared to the native Android system, CReDroid can have a fine-grained control over the applications' permission usage behavior, and thus effectively solve the problem the abuse of permission and intra-app malicious code attack. Experimental results show that CReDroid introduces a small load consumption and can better protect users' privacy.

6. Acknowledgments

This work was partially funded by the National Natural Science Foundation of China (General Program) under Grant No. 61572253 and the National Postdoctoral Fund under Grant No. 2011M500124.

References

[1] IDC. Smartphone OS Market Share, 2016 Q2 [EB/OL]. http://www.idc.com/prodserv/smartphone-os-market-share.jsp.
[2] FELT A P, CHIN E, HANNA S, et al. Android permissions demystified; proceedings of the ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, Usa, October, F, 2011 [C].
[3] NAUMAN M, KHAN S and ZHANG X, "Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints", proceedings of the ACM Symposium on Information, Computer and Communications Security, F, 2010.
[4] XU B, ZHANG Y and YANG M, "GRANTDROID: A METHOD SUPPORTING IN-CONTEXT PERMISSION GRANTING FOR ANDROID", Computer Applications & Software, 2014.
[5] JUNG C, FETH D and SEISE C, "Context-Aware Policy Enforcement for Android", proceedings of the IEEE International Conference on Software Security and Reliability, F, 2013.
[6] Wentang Li, Fan Jiang and Wei Sun, "The Method and Realization of Android Applications Malicious Code Static Injection", Journal of Information Security Research, 2016.
[7] Backus-Naur Form [EB/OL]. https://en.wikipedia.org/wiki/Backus-Naur_form.
[8] BOND M D, MCKINLEY K S. Probabilistic calling context [J]. Acm Sigplan Notices, 2007, 42(10): 97-112.
[9] Arp Daniel, Spreitzenbarth Michael, Hubner Malte, et al.DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket [C]; Network and Distributed System Security Symposium (NDSS), 2014.
[10] Chaurasia P. Dynamic analysis of Android malware using DroidBox[J]. Dissertations & Theses - Gradworks, 2015.
[11] Enck W, Gilbert P, Han S, et al. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones[C]// Usenix Conference on Operating Systems Design & Implementation. 2014:393-407.
[12] ApkTool [EB/OL]. 2016. https://github.com/iBotPeaches/Apktool.