Your phone is leaking data! Evaluating Android content provider permissions

Prajit Kumar Das
University of Maryland, Baltimore County
prajit1@umbc.edu

Sandeep Narayanan
University of Maryland, Baltimore County
sand7@umbc.edu

Stanislav Bobovych
University of Maryland, Baltimore County
stan@umbc.edu

Anupam Joshi
University of Maryland, Baltimore County
joshi@umbc.edu

Nilanjan Banerjee
University of Maryland, Baltimore County
nilanb@umbc.edu

Ryan Robucci
University of Maryland, Baltimore County
robucci@umbc.edu

ABSTRACT

The number of mobile devices in the world surpassed the number of personal computers in 2010. Mobile devices now carry sensitive personal data, captured through sensors on the phone, as well as confidential corporate data through work emails and apps. As a result, they have become lucrative targets for attackers and the privacy and security of these devices have become a vital issue. Existing access control mechanisms on these devices, which mostly rely on a one-time permission grant, are too restrictive and inadequate. Such mechanisms are incapable of controlling contextual or custom app-data flows. In this paper we focus on this scenario and show how data leakages may occur due to developer inadequacy and a lack of proper checks for such leakages. We describe a potential loophole in the Android permission verification mechanism and a way to capture such a vulnerability on a user's mobile device. We also show a mechanism of injecting such a vulnerability into any app.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Access Controls*

Keywords

Access Control, Android Content Providers, Permission Control

1. INTRODUCTION

Mobile devices have become ubiquitous due to their power, convenience and low cost and Android has become the biggest player in the market. The latest reports from Google boast of more than a billion 30-day active user [8]. According to the International Data Corporation's Worldwide Quarterly Mobile Phone Tracker report, Android has a 85% market share in the smartphone category. Apps from the Google Play Store and a variety of other outlets like Amazon App Store and Samsung Galaxy Apps provide a plethora of ways through which Android users can get their apps [1]. According to Statista [7], as of July 2015, there are more than 1.6 million Android apps in the Google Play Store.

The proliferation of smartphones has led to the popularity of the BYOD (Bring-Your-Own-Device) paradigm, whereby people use their personal devices in their workplaces to access business information and services. Naturally, this creates a greater need to ensure strong access control mechanisms for the data on such devices. In certain domains the access control needs are critical. For example, for Medical and Health and Fitness apps it is essential to maintain the highest level of security for user data and, if being used by providers, patient data. Hospitals today use various hardware devices that are smart enough to communicate with smartphones and may even contain sensitive medical data. In addition, Android apps are capable of collecting a huge amounts of data about the smartphone users, often without their knowledge.

In this paper, we introduce Heimdall [1], a heuristics based system that is currently in-the-works in our group. Heimdall is capable of detecting common vulnerabilities on an Android device that can cause leakage of app data. Heimdall has been created with a BYOD scenario in mind, where part of the system resides on the mobile device and part on the server. The server-side includes a dashboard that gets notifications of apps being installed on the mobile devices used by the employees of the organization. The system is then able to analyze and detect if the app is vulnerable with respect to a list of previously known heuristics. We are adding new heuristics as we discover and study them.

In the current paper, we have focused on vulnerability in custom permissions created by app developers. These permissions exist to protect the app developers' data available through their own content providers. It is advised by Google that, if an app developer creates a content provider for allowing access to there own data, they should also create a permission to control access to it. However, this requirement

[1]Heimdall is the all-seeing and all-hearing guardian sentry of Asgard who stands on the rainbow bridge Bifröst to watch for any attacks on Asgard

HotMobile'16 February 23-24, 2016, St. Augustine, Florida, USA

is not a stringent one and one might simply ignore creating such a permission. We show in this paper how such a vulnerability might lead to leakage of app data. We use two different mechanisms to demonstrate the issue. We show that it is possible to exploit this vulnerability using our own data access app and content provider app pair. We also show that it's possible to reverse engineer and repackage any standard app to create this vulnerability. We did observe that it is possible to check for this issue in your code instead of delegating this issue to Android but through our evaluation we show that such a check might be beyond standard practices.

Previous work points out the extensive research that has gone into various mechanisms to study vulnerabilities in Android apps. The mechanisms have ranged from app metadata analysis by Pandita et. al. [6], to detecting malware by studying their characteristics like installation methods, activation mechanisms and malicious payload nature by Zhou et.al. [9]. Such studies indicate a need for better mobile anti-malware solutions and access control mechanisms. We can understand from the extensive work done that there is significant knowledge about vulnerabilities on Android and ways to detect them. In this paper we present Heimdall, a system which can detect such vulnerabilities, and show an example of how one of these vulnerabilities can be detected using our system.

The rest of the paper is organized as follows. We describe our system overview in the section 2. That is followed by a description of the problem at hand in section 3. We also present a way such a loophole can be introduced in any Android app in this section. We present a working prototype that is capable of detecting such a vulnerability in section 4. We conclude the paper with a discussion of related work in section 5 and future research directions that can lead to more vulnerability discoveries in section 6.

2. SYSTEM OVERVIEW

Heimdall has two components: the first is an app installed on a user's mobile device and the second is a Web service that receives install, uninstall and update notifications when these events occur on the device. Upon notification, the server processes all heuristics that apply to the app and generates a set of actions for a system administrator. At this point the system administrator can take an appropriate action based on the detected threat level. Our present prototype, includes a simple content provider heuristic that estimates the impact of a vulnerability. The only solution that is possible, at the moment, is to uninstall the app and that notification is then sent back to the user's device automatically.

Heimdall server has two additional capabilities. The first is to generate reverse engineered apps that we then test on the mobile devices. The reverse engineering process removes any provider associated permission and ensures that the "exported" tag for the provider is set to true. The second capability is to detect missing provider permissions for known apps. For demonstrating these capabilities, we downloaded about 1500 apps from the Google Play Store. We used a tool called apktool [2] to decompile the Android binary application packages (apks) and parse the manifest files to detect the presence of such a vulnerability in the apps. Naturally,

[2]A tool for reverse engineering 3rd party, closed, binary Android apps `https://ibotpeaches.github.io/Apktool/`

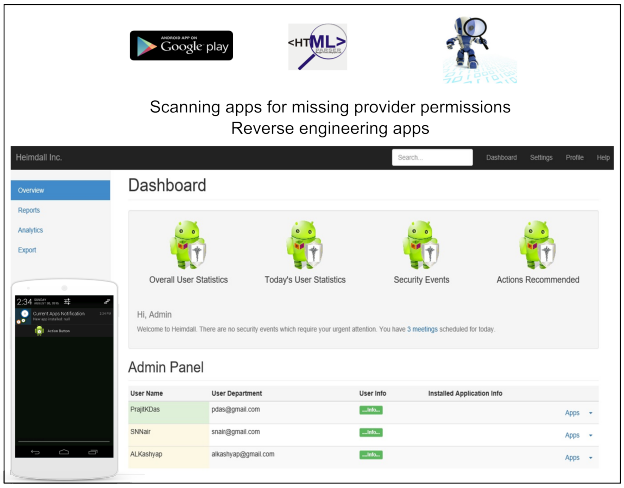

Figure 1: System Overview

as we include more heuristics, Heimdall will become capable of detecting more such vulnerabilities.

3. VULNERABILITY DESCRIPTION

When it comes to content providers, Google's Android documentation describes two possible scenarios. The first states that data from a provider that specifies no permissions should not be accessible from other apps.

- "[...] If a provider's application doesn't specify any permissions, then other applications have no access to the provider's data. However, components in the provider's application always have full read and write access, regardless of the specified permissions." [3]
- "[...] All applications can read from or write to your provider, even if the underlying data is private, because by default your provider does not have permissions set. To change this, set permissions for your provider in your manifest file, using attributes or child elements of the <provider> element. You can set permissions that apply to the entire provider, or to certain tables, or even to certain records, or all three." [4]

Unfortunately, we found the first statement to be untrue. Table 3 lists the various scenarios and points out when a content provider app is not associated with a permission we may have data leakage. This happens because Android does not verify that each provider has an associated permission. There is one more condition required for this vulnerability to open up the provider to potential attacks and that is the exported setting to be set as true in the Manifest file for the provider app, as shown in code Listing 1. Possible solutions to mitigate this issue would require, either a change in how Android handles content provider access control or a change in the app developer's code.

[3]Error in specification `http://developer.android.com/guide/topics/providers/content-provider-basics.html#Permissions`

[4]Correct specification `http://developer.android.com/guide/topics/providers/content-provider-creating.html#Permissions`

Content Provider app	Content accessing app	Remark
No permission associated with provider	No permission used	**Potential data leakage**
Permission associated with provider	No permission used	Permission denied
Permission associated with provider	Permission used	Ideal scenario
No permission associated with provider	Permission used	No error

Table 1: Scenario when data leakage may happen

Listing 1: Provider exported tag set as true

```
...
<provider
        android:name=''contentProviderName''
        android:authorities=''authorityName''
        android:exported=''true''>
...
```

Such a vulnerability can also be created deliberately. As shown in the work by Zhou et. al. [9] app repackaging is one of the most common techniques for android malware creation, we show that it is possible through a simple change in code to introduce such a vulnerability in any app. There are some obvious ways to check for such manipulations and top developers in the Google Play Store usually do include such checks. However, these checks are not part of the android framework or operating systems and therefore a repackaged app can be used to fool users into installing a rogue application and allow their data to be stolen.

4. EVALUATION

We have discussed a potential loophole in android's custom provider data flow, in this paper. In order to demonstrate this problem we built a proof-of-concept(PoC). All our experiments were ran on a LG Nexus 5 device with Android Marshmallow 6.0 installed on it. In our PoC, we have an app(COMMAND) that has an exported content provider. We created another app(Parser) that is capable of accessing the content provider. We use two different application package sets and observe the results. The first set contains the COMMAND app without any permission specification and Parser app without any permission request. The second set contains the COMMAND app with permission specification and association with the provider that was created. It also includes the Parser app with the request for the permission created by COMMAND.

- Case 1: COMMAND has associated permission, Parser has requested said permission. We see in Figure 2 that, in this case there are no errors and we are able to make a sample query to the content provider.
- Case 2: COMMAND has no associated permission, Parser has not requested said permission. We see in Figure 3 that, in this case there are no errors and we are again able to make a sample query to the content provider. We propose that there should a check in such a case to ensure that data access is to be allowed or denied. At present this does not happen and app developers resort to individual techniques to protect their data.
- Case 3: COMMAND has associated permission, Parser has not requested said permission. We see in Figure 4 that, causes a permission denial error which is what we expected.
- Case 4: COMMAND has no associated permission, Parser has requested an unknown permission. There is no error in this case on the phone. We can ignore this error because this does not cause any leakage from the data provider perspective.

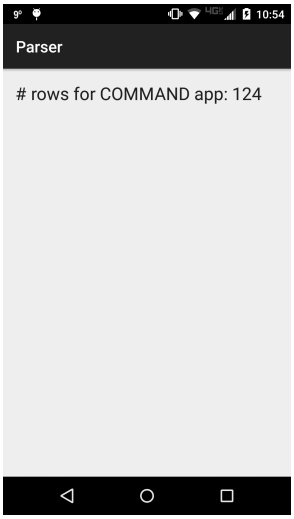

Figure 2: Android content provider accessed with permission

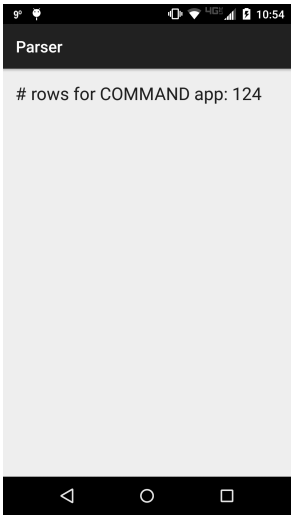

Figure 3: Android content provider accessed without permission

The PoC proves that there is no difference from user perspective between an app which has a content provider with proper protection using appropriate permission and an app which does not have such access control implemented. Therefore, user data can potentially leak without user knowledge.

We also ran our analysis on a set of 1500 randomly selected applications with a mix of popular applications like Facebook, GMail, Instagram as well as less popular and unknown apps like Expense Manager, Call App etc. Our system found 150 applications with provider set as exported and no associated permission for the provider. Therefore about 10% of apps have this potential loophole but we wanted to find out if we could change an app to leak it's data. This led to our second set of experiments in trying to determine if these had incorporated additional protection apart from the standard android permission mechanisms. For these experiments we used the Facebook app and the Google Fit app. We removed all permissions associated with the providers on both the apps. We also set all the providers' exported setting to true. When these modified apps were installed on our test phone the Google fit app immediately crashed. We used logcat, the Android logging system that provides a mechanism for collecting and viewing system debug output,

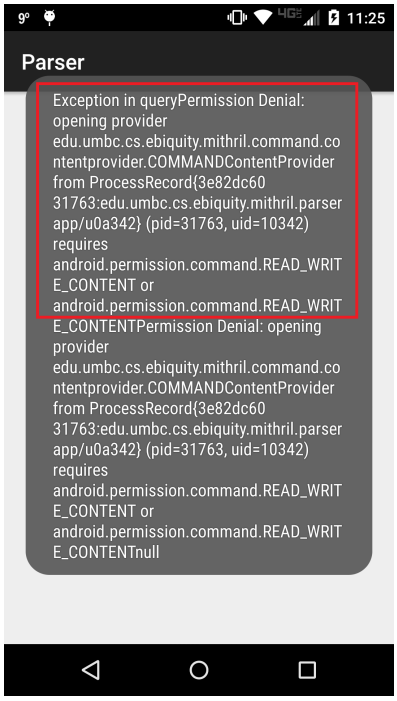

Figure 4: Android content provider permission denial

```
Process: com.google.android.apps.fitness, PID: 5528
java.lang.SecurityException: Caller isn't signed with recognized ke ⏎
ys!
at android.os.Parcel.readException(Parcel.java:1599)
at android.database.DatabaseUtils.readExceptionFromParcel(Database ⏎
Utils.java:183)
at android.database.DatabaseUtils.readExceptionFromParcel(Database ⏎
Utils.java:135)
at android.content.ContentProviderProxy.call(ContentProviderNative ⏎
.java:646)
at android.content.ContentProviderClient.call(ContentProviderClien ⏎
```

Figure 5: Google Fit app checks for certificate signatures

to observe what was happening on the phone and the cause of the errors. For Google Fit we found that there is an additional check on the key signature that the app is carrying out and it simply crashes because the signature is detected as unknown. You can see the error in Figure 5.

On the other hand the Facebook app does not ever crash and works like a normal Facebook app. However, when we try to access the app's content provider it blocks our attempt and we can see in Figure 6 that it controls access to it's own component. Therefore, app developers are clearly detecting such issues on their apps but not always. We found at least one app that we were able to get access to from our randomly selected apps collection. This app is an Expense Management app and when we tried to access it's content provider we got no checkpoints or errors. Although, as seen in Figure 7, our query failed but that was because the app wasn't writing it's data to it's database.

We are currently processing more apps to find out if they include such checks as encoded by popular apps from Google or Facebook. This processing takes a long time as because

```
java.lang.SecurityException: Component access not allowed.
at com.facebook.content.PermissionChecks.a(PermissionChecks.java:1 ⏎
63)
at com.facebook.content.AbstractContentProvider.c(AbstractContentP ⏎
rovider.java:243)
at com.facebook.content.AbstractContentProvider.d(AbstractContentP ⏎
rovider.java:249)
at com.facebook.content.AbstractContentProvider.query(AbstractCont ⏎
entProvider.java:410)
at android.content.ContentProvider$Transport.query(ContentProvider ⏎
.java:238)
at android.content.ContentProviderNative.onTransact(ContentProvide ⏎
rNative.java:112)
at android.os.Binder.execTransact(Binder.java:453)
```

Figure 6: Facebook checks controls access to it's component

we have to manually find the databases on the phone using a rooted phone and a SQLite explorer app. Thereafter, we have to make guesses for patterns that apps might have used in their content provider code. There are commonly used patterns like '#' that can be used to get access to the data and we are trying to use them to find out apps which have such a vulnerability.

5. RELATED WORK

There have been multiple attempts at achieving the goal of properly managing access control on mobile (Android) devices. Efforts have been made by the open source community through the XPrivacy project (needs a rooted phone), the Privacy Guard project (available on Cyanogenmod, a custom Android ROM), the PDroid application (needs a rooted device). Research project by Conti et. al. [2] (CRePe), Enck et al. [3] (TaintDroid) and Jagtap et al. [5] (Preserving Privacy in Context-Aware Systems) have made similar efforts. CRePE described a system where security policy enforcement was carried out based on context of the smart phone. TaintDroid was a research effort where the data flow on an Android device was studied to figure out when sensitive data left the system via an untrusted application. The work of Jagtap et al. [5] focused on constraining data flow in a context-aware system using a policy-based framework. A related work by Ghosh et al. [4] used a similar policy driven approach to constrain application permissions based on context. Playdrone : Crawls Playstore- how playstore evolved - source code analysis of library usage - similar app detection-secret authentication key storage (can be found by decompilation) (1) native libraries are heavily used by popular Android applications, limiting the benefits of Java portability and the ability of Android server overloading systems to run these applications, (2) 25% of Google Play is duplicative application content, and (3) Android applications contain thousands of leaked secret authentication keys which

Andradar : First, we can discover malicious applications in alternative markets, second, we can expose app distribution strategies used by malware developers, and third, we can monitor how different markets react to new malware. To identify and track malicious apps still available in a number of alternative app markets.

Android Security : discuss the Android security enforcement mechanisms, threats to the existing security enforcements and related issues, malware growth timeline between

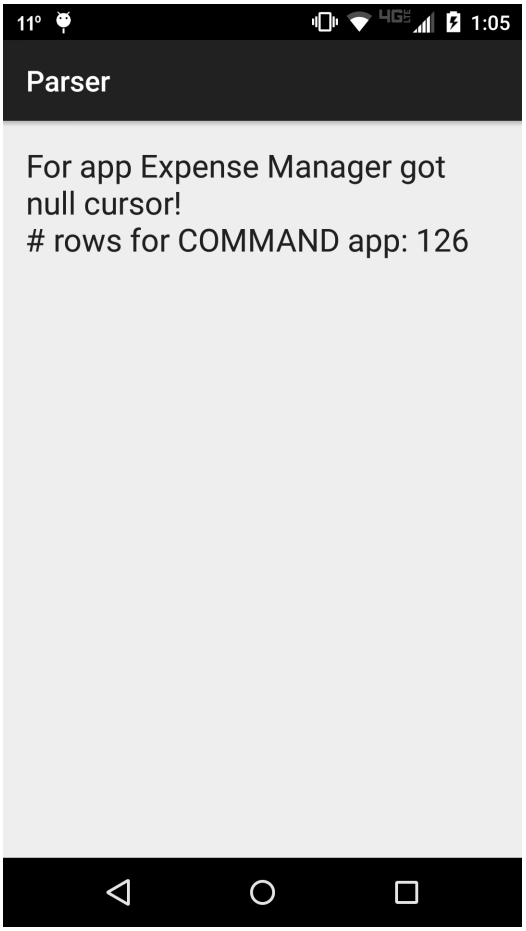

Figure 7: No check points were found on a less popular app

2010 and 2014, and stealth techniques employed by the malware authors, in addition to the existing detection methods. This review gives an insight into the strengths and shortcomings of the known research methodologies and provides a platform, to the researchers and practitioners, toward proposing the next-generation Android security, analysis, and malware detection techniques.

-ANDRUBIS:, a fully automated, publicly available and comprehensive analysis system for Android apps. ANDRUBIS combines static analysis with dynamic analysis on both Dalvik VM and system level, as well as several stimulation techniques to increase code coverage.

changes in the malware threat landscape and trends amongst goodware developers. Dynamic code loading, previously used as an indicator for malicious behavior, is especially gaining popularity amongst goodware App analysis for astma!!

App behavior against description CHABADA tool clustering apps by description topics, and identifying outliers by API usage within each cluster, our CHABADA approach effectively identifies applications whose behavior would be unexpected given their description. Recommendations for the Android ecosystem

author et.al. developed a formal Android permission model to analyze the permission protocol used using Alloy. Using Alloy analyzer, they reasoned over the model to detect possible vulnerabilities in the protocol. It also generated counter examples which can possibly exploit the vulnerabilities in the protocol. Their work could detect a vulnerability in which an application with normal security level permission was able to access another application's dangerous level custom permission given the following conditions. First, both the permission names are the same and second the application with lesser security level is installed first. But we differentiate from them such that instead on just focusing on

6. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a Heimdall, a system that is capable of detecting security vulnerabilities on a mobile device. We use a heuristics based approach to detect these vulnerabilities. We also discussed a potential loophole in Android's custom content provider's security. This loophole could allow a malicious app to steal users' data from their phones. We built a proof-of-concept for demonstrating this loophole and we also tested a random sample of 1500 apps to find out if this vulnerability exists in them. We reverse engineered two popular apps and a non-popular app, to see, if there are additional checks present in the code, to handle access control to the data. We presented our observations from that process which led us to conclude that such checks are possible and present in some apps but not all apps. In conclusion we can say that this is a potential loophole that could lead to user data leakage and thus have serious implications. Therefore, there needs to be some checking mechanism in form of an API to verify app signature keys or to verify component access control or maybe even strict permission association for custom providers.

In the future, we hope to include more heuristics in Heimdall and capture more such vulnerabilities. As we have discussed in the related work section, a lot of research has been undertaken in this area and we hope to incorporate the ability to detect these vulnerabilities in Heimdall. As the Heimdall project's primary goal is to be deployed in a BYOD scenario we are working on a mechanism to actually control the data flow on android. This will allow us to study what data is being transferred to and from the phone as well as implement policies defined by the system administrator. Detecting discrepancy between app's expected behavior and actual behavior is also being studied by many researcher but we feel this problem still remains unsolved and we hope to make that into a feature of Heimdall.

7. ACKNOWLEDGMENTS

Support for this work was provided by NSF grants 0910838 and 1228198.

8. ADDITIONAL AUTHORS

Additional authors: Ting Zhu (University of Maryland, Baltimore County, email: `zt@umbc.edu`) and Tim Finin (University of Maryland, Baltimore County, email: `finin@umbc.edu`).

9. REFERENCES

[1] R. Chang. 10 alternative android app stores, 2014.
[2] M. Conti, V. T. N. Nguyen, and B. Crispo. Crepe: Context-related policy enforcement for android. In M. Burmester, G. Tsudik, S. Magliveras, and I. Ilic,

editors, *Information Security*, volume 6531 of *Lecture Notes in Computer Science*, pages 331–345. Springer Berlin Heidelberg, 2011.

[3] W. Enck, P. Gilbert, B.-G. Chun, L. P., Cox, J. Jung, P. McDaniel, and A. N. Sheth. 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*, pages 1–6, 2010.

[4] D. Ghosh, A. Joshi, T. Finin, and P. Jagtap. Privacy control in smart phones using semantically rich reasoning and context modeling. In *Security and Privacy Workshops (SPW), 2012 IEEE Symposium on*, pages 82–85, 2012.

[5] P. Jagtap, A. Joshi, T. Finin, and L. Zavala. Preserving privacy in context-aware systems. In *Fifth IEEE Int. Conf. on Semantic Computing (ICSC)*, 2011.

[6] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. Whyper: Towards automating risk assessment of mobile applications. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 527–542, Berkeley, CA, USA, 2013. USENIX Association.

[7] Statista. Number of available applications in the google play store from december 2009 to july 2015, 2015.

[8] C. Trout. Android still the dominant mobile os with 1 billion active users, 2014.

[9] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109, May 2012.