M-Perm: A Lightweight Detector for Android Permission Gaps

Conference Paper · May 2017

DOI: 10.1109/MOBILESoft.2017.23

4 authors, including:

Mohamed Wiem Mkaouer
Rochester Institute of Technology
26 PUBLICATIONS **186** CITATIONS

2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)

M-Perm: A Lightweight Detector for Android Permission Gaps

Piper Chester, Chris Jones, Mohamed Wiem Mkaouer and Daniel E. Krutz
Department of Software Engineering
Rochester Institute of Technology
Rochester, NY, USA
Email: {pwc1203,caj6831,mwmvse,dxkvse}@rit.edu

***Abstract*—Android apps operate under a permissions-based system where access to specific APIs are restricted through the use of permissions. Unfortunately, there is no built-in verification system to ensure that apps do not request too many or too few permissions, which could lead to serious quality and/or privacy concerns. Apps requesting too many permissions create unnecessary vulnerabilities, leaving the potential for abuse by SDKs within the app or other malicious apps installed on the device. In order to assist with the discovery of misused permissions, we created a new detection tool, *M-Perm*, which combines static and dynamic analysis in a computationally efficient manner compared to existing tools. M-Perm also identifies permission usage in apps including requested normal, dangerous, and 3rd party permissions. The tool, complete usage instructions, and screencast are available online: http://www.m-perm.com**

I. Introduction

A cornerstone of Android's security is its use of permissions. Unfortunately, there is no built-in verification system to ensure that apps adhere to the *principle of least privilege* and do not request unnecessary permissions, which increases the app's attack surface and make it more susceptible to a variety of security and privacy related issues [3]. Several studies have analyzed app permissions through combining static and dynamic analysis. Stowaway [3] introduced the idea of mapping between all the apps interactions with the Android system. PScout [1] uses a Class Hierarchy Analysis (CHA) of a decompiled source code to generate the list of all objects and uses the Soot library to link objects to their call graphs. This allows detecting any implicit use of permissions by any of the app's components. Similarly, Spark-Android [2] extends PScout's static analysis by using CHA and field-sensitive analysis. Table I provides an overview of these tools.

A limitation of Stowaway is the need of test cases that maximize the coverage of the instrumented app's code in order to detect permissions, another limitation of using this dynamic analysis is the detection of only overprivileges. PScout and Spark-Android were designed around static analysis to overcome the limitations of Stowaway. Still, besides using string analysis to detect any explicit permission use, e. g., calling permissions check methods of *Android.content.Context*, using the Soot[1] library they also generate the call graph of both the app and the system to detect implicit permissions. Although this analysis does reveal permission issues, it is highly dependent on the tested Android release and the tools do not provide any automated support on how to re-analyze any new release of Android. Additionally, they are not tailored for the permission system used in Android API $\geq$ 23. To address these limitations, we have created a new tool *M-Perm* that combines string analysis with static analysis to detect permissions misuse. In contrast with PScout and Spark-Android, M-Perm accepts but does not require any prior built and refined call graph.

[1]https://sable.github.io/soot/

II. M-Perm Tool

The primary components of M-Perm are shown in Figure 1. The Driver is responsible for interacting with the configuration file and the decompilation tools that reverse engineer the APK files. M-Perm decompiles APK files into source code using a three-step process encompassing three popular tools including Apktool[2], dex2jar[3], and JD-Core-java[4]. Although there are several other available tools for reverse engineering apps, we found that our selected tools were able to quickly and reliably reverse engineer apps while adequately being able to overcome many of the challenges associated with obfuscated apps. PMD[5] is ran on the decompiled source files to eliminate dead code from the analysis and minimize the generated graph.

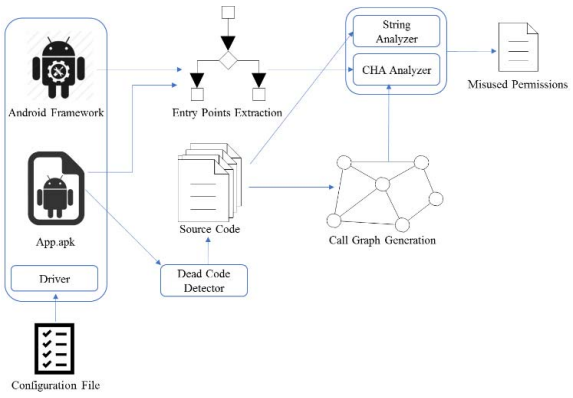

Fig. 1: M-Perm Architecture

[2]https://ibotpeaches.github.io/Apktool/
[3]https://sourceforge.net/projects/dex2jar/
[4]https://github.com/nviennot/jd-core-java
[5]https://pmd.github.io/

978-1-5386-2669-6/17 $31.00 © 2017 IEEE
DOI 10.1109/MOBILESoft.2017.23

TABLE I: Overview of Existing Tools

Tool	**M**ethodology	**L**ast Compatible Android Release	**P**ublicly Available
Stowaway	API-calls test case generation to detect permissions	Android 2.2.2	
PScout	CHA	Android 5.1.1	Yes
Spark-Android	CHA + Spark for call graph refinement	Android 4.0.1	

Static analysis is performed after inputting the system's and app's bytecodes to the Soot library. The permission gap is then generated using similar steps as defined by Bartel et al. [2]. M-Perm first scans the permissions declared in the manifest files, those permissions are saved in a vector labeled $P_{manifest}$ whose size represents the total number of permissions which can be declared. Every dimension in $P_{manifest}$ is set to 0 if the permission is not declared in the file and 1 otherwise. Similarly, we compute the vector P_{src} with every dimension in P_{src} is set to 0 if the permission is not explicitly solicited by any of the source code methods and 1 otherwise. Secondly, we use the app's call graph to analyze the reachability of the methods to the framework's entry points. This allows the generation of a vector labeled EP_{active}, whose size represents the total number of entry points, every dimension in EP_{active} is set to 1 if it exists any afferent or efferent coupling between a method and an entry point and 0 otherwise. Since the entry points may lead accessing resources protected by permissions, the matrix EP_{perm} has as rows the number of entry points and as columns the number of permissions. For a given cell $EP_{perm(i,j)}$, it is set to 1 if entry point i has accessed to a resource protected by a permission j. So the permissions being currently used by the app, are stored in a vector labeled $P_{checked}$ and computed as follows: $P_{checked} = P_{src}$ OR (EP_{active} AND EP_{perm}) The under and overprivileges are calculated as: $P_{under} = P_{checked} - P_{manifest}$ $P_{over} = P_{manifest} - P_{checked}$

M-Perm's output is placed in two output files. The first is `analysis_<package_name>` which lists the normal, dangerous[6], 3rd party, and under & over privileges found in the app. The file also correlates the requested dangerous permissions with their associated permission groups, along with all dangerous permissions in the app and how they are being used. For example, dangerous permissions may be directly used by the app in a function, or the app may merely check to see if the app already has access to the permission. M-Perm will provide the context of how the app is using this permission. Knowing the context of permission requests can be useful to a variety of researchers including those analyzing app permission leaks [4], or those simply looking to better understand app permissions and how they are used. M-Perm has been designed to use an easily interchangeable configuration file to make using different versions of Android permissions as easy as possible. This lack of interchangeability has caused many leading Android permission analysis tools to lag and even become quickly outdated. M-Perm's analyzers are independent, the developer can run the syntactic analysis without referring to the call graphs, in case the system's call graph is not available.

[6]https://developer.android.com/guide/topics/permissions

The file `source_report_<package_name>` describes the usage of permissions inside of the app. The report lists the files that permission requests occur in, along with the block of code that is requesting the permission.

III. Evaluation

To assess the effectiveness of M-Perm, we investigated the existence of a permission gap on Android Marshmallow apps from both Google Play[7] and apkMirror[8]. A few of these apps included Uber, Skype, LinkedIn, Twitter, and Google Calendar. The results of this analysis are shown in Table II.

TABLE II: Analysis Results for 50 Apps

	Total	**avg/App**
Total Permissions	1119	22.4
Dangerous Permissions	531	10.6
Third Party Permissions	321	6.4
Under Privileges	89	1.8
Over Privileges	110	2.2

Recent research [2] has found that 11% of around 500 tested apps extracted from Google Play contain at least one overprivilege. We found that 2.2% of apps contained an overprivilege, this was expected since the analyzed apps are popular well-designed projects, and that the updated permission model in API $\geq$ 23 may affect permission misuse. However, our sample size is very small and calls for future work in this area. Unfortunately, with the absence of datasets used by the state of art tools, we couldn't directly compare our findings with their results. Moreover, comparing with the existing tools requires not only using older Android releases but also locating previous releases of apps that may no longer be available. As part of the future work, we plan on extending the static analysis by adding the Spark refinement introduced by Bartel et al. [2].

References

[1] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM, 2012.

[2] A. Bartel, J. Klein, M. Monperrus, and Y. Le Traon. Static analysis for extracting permission checks of a large scale framework: The challenges and solutions for analyzing android. *IEEE Transactions on Software Engineering*, 40(6):617–632, 2014.

[3] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638. ACM, 2011.

[4] D. Wang, H. Yao, Y. Li, H. Jin, D. Zou, and R. H. Deng. Cicc: a fine-grained, semantic-aware, and transparent approach to preventing permission leaks for android permission managers. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, page 6. ACM, 2015.

[7]https://play.google.com/

[8]https://www.apkmirror.com