Unintentional Bugs to Vulnerability Mapping in Android Applications

Garima Bajwa, Mohamed Fazeen, Ram Dantu and Sonal Tanpure
Department of Computer Science & Engineering
University of North Texas Denton, Texas 76207
(garimabajwa, mohamedfazeen)@my.unt.edu, rdantu@unt.edu and tanpure.sonal@gmail.com

***Abstract*—The intention of an Android application, determined by the source code analysis is used to identify potential maliciousness in that application (app). Similarly, it is possible to analyze the unintentional behaviors of an app to identify and reduce the window of vulnerabilities. Unintentional behaviors of an app can be any developmental loopholes such as software bugs overlooked by a developer or introduced by an adversary intentionally. FindBugs^{TM}and Android Lint are a couple of tools that can detect such bugs easily. A software bug can cause many security vulnerabilities (known or unknown) and vice-versa, thus, creating a many-to-many mapping. In our approach, we construct a matrix of mapping between the bugs and the potential vulnerabilities. A software bug detection tool is used to identify a list of bugs and create an empirical list of the vulnerabilities in an app. The many-to-many mapping matrix is obtained by two approaches - severity mapping and probability mapping. These mappings can be used as tools to measure the unknown vulnerabilities and their strength. We believe our study is the first of its kind and it can enhance the security of Android apps in their development phase itself. Also, the reverse mapping matrix (vulnerabilities to bugs) could be used to improve the accuracy of malware detection in Android apps.**

***Keywords*—android applications; bugs; mapping; security; vulnerability**

I. Introduction

There has been an unprecedented growth in the use of smartphones across the globe. In January 2015 Android ranked as the top smartphone platform in the U.S with 53.1 percent market share, followed by Apple with 41.6 percent, BlackBerry with 1.8 percent, Microsoft with 3.4 percent and Symbian with 0.1 percent [1]. Android is based on Linux kernel where applications run data independently and inter process communication is strictly based on a permission system. Application download requires users to *blindly grant access to the listed permissions or deny installation*. Peter Bright's article[1] describes "Google has no ability to push out updates to the operating system; it has to depend on a range of OEMs and network operators to adopt its source code changes and distribute them to users. Both Apple and Microsoft, in contrast, have a direct channel to update their mobile operating systems". This limitation along with an open source platform allows adversaries to take full advantage of the Android ecosystem that compromises integrity, availability and confidentiality of the user.

Applications available online in Android market are pre-screened by Google corporation but attackers bury malware codes within an app which don't infest until after the app has been downloaded. Malicious intents have been well studied and developer tools exist to discover and correct them by static or dynamic analysis on the source code of an application. However, little has been done to address the vulnerabilities caused by unintentional software defects.

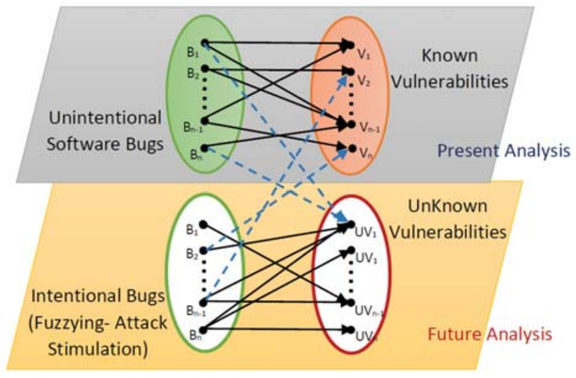

Fig. 1. The proposed system of bugs to vulnerability mapping

Given the rise in mobile malwares, we intend to identify the vulnerabilities exposed by the unintentional software bugs in the application. These can be used by the cyber exploiters as a gate pass to the resources and sensitive information of the mobile system. There have been numerous studies on bugs and vulnerabilities [2], [3], [4], [5], [6], [7], [8] but no work has systematically explored the mapping of bugs to vulnerabilities and the consequent threats in Android ecosystem. Such mappings will enhance the identification of malicious Android applications. Our empirical study is the first of its kind and the preliminary results elucidate the need for more sophisticated mapping between bugs, vulnerabilities and threats. The proposed system of mapping is shown in Figure 1.

II. Data Collection

The first requirement to formalize the mapping concept was to prepare a repository of android applications with their original source codes i.e. which were *not reverse engineered* using tools like APK tool, JD-GUI and DEX2JAR [2]. Therefore, a set of 230 applications was downloaded from the "Github"[3] web site. The applications were grouped in two ways based

[1]http://arstechnica.com/security/2015/01/google-wont-fix-bug-hitting-60-percent-of-android-phones/

[2]https://code.google.com/p/android-apktool/, https://code.google.com/p/dex2jar/, http://jd.benow.ca/
[3]https://github.com/

on the functional categories or the permissions they require to execute a service. Developers can set certain permission attributes to require authorization to access the resources of the mobile system for the app's functioning. These permissions are defined in an XML file called manifest "AndroidManifest.xml". To ensure the selected applications were popular and common, the application list was sorted based on popularity and the first 230 were selected. Further, these 230 apps were classified based on four major permissions and six functional categories as show in Tables 1 and 2. The next sections discuss the methods of implementation of our continuous learning mapping model.

Functional Category	Number of Apps
General	5
Finance	5
Gaming	5
Weather	5
Online Shopping	5
Weather	5
Total	30

TABLE I. APPLICATIONS GROUPING BASED ON FUNCTIONAL CATEGORIES

Major Permission Type	Number of Apps
Internet	50
Location access	50
Storage	50
Short Message Service (SMS)	50
Total	200

TABLE II. APPLICATIONS GROUPING BASED ON PERMISSION CATEGORIES

III. Bugs - Vulnerability Mapping Procedure

In this approach, two static code analysis tools were used. They are Lint and FindBugs [9], [10], [11]. The outcome of static analysis provided the list of bugs found in each app with its severity level, called the bug rank. Bugs are given a rank 1-20, and grouped into the categories scariest (rank 1-4), scary (rank 5-9), troubling (rank 10-14), and of concern (rank 15-20). The list of vulnerabilities included in this study were obtained from existing known vulnerabilities [12], [13], [6]. The empirical mapping of bugs to expected vulnerabilities was generated as a 2D matrix where each column represents a vulnerability and each row a bug. Each of these bugs were manually analyzed using the bug description in FindBugs and predicted into a vulnerability that might occur while executing an app. The architecture of the model is illustrated in the Figure 2. The construction of the matrix cell was done by two methods namely Severity mapping and Probability Mapping. A proposed third method combining the two methods is addressed as future work, shown in Figure 3.

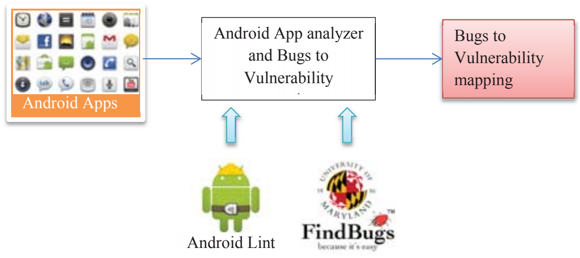
Android Apps
Android App analyzer and Bugs to Vulnerability
Bugs to Vulnerability mapping
Android Lint
FindBugs

Fig. 2. Model for generating the matrix of Bugs to vulnerability mapping

A. Severity Mapping

In this technique, we used the functional category containing six groups and 30 applications as shown in Table I. Each category of 5 applications was then subjected to FindBugs analysis to obtain a potential list of bugs with their ranks. The known vulnerability was assigned a boolean value of zero or one in the static vulnerability list using the description of a bug in FindBugs report. The severity of a bug-vulnerability was calculated as Low (1-7), Medium (8-14) or High (14-20) by taking the median of all the ranks obtained in each category.

B. Probability Mapping

This method was similar to the previous one, only differing in the grouping category and number of tested applications. Application grouping of permission category, containing four groups and 200 applications were used as shown in Table II. Each category of 50 applications was again subjected to FindBugs analysis to obtain a potential list of bugs with their ranks. After assigning a boolean value to the known vulnerability in static vulnerability list, the probability of occurrence of a bug-vulnerability was calculated by using the following formulas.

$$P(B) = \frac{\textit{No of Apps having the bug}}{\textit{Total No of Apps in dataset} \times \textit{Total No of Bugs}} \tag{1}$$

$$P(V) = \sum_{i=0}^{N_b} P(V_i|B_i) \times P(B_i) \tag{2}$$

where N_b is categories of bugs in data set

$$P(V|B) = \frac{\textit{No of Apps having the vulnerability for a given bug}}{\textit{Total No of Apps in the dataset}} \tag{3}$$

$$P(B|V) = \frac{P(V|B) \times P(B)}{P(V)} \tag{4}$$

C. Mixed Mapping

The mixed mapping procedure can be used to first group the applications based on functional categories and then sub-group them into permission categories as shown in Figure 3. This approach would help to narrow down the range of known vulnerabilities and aid in investigating more sophisticated mapping matrix by developing an Index which is a function of both severity and probability mapping indexes as shown below;

$$I(B_i, V_i) = f(S, P(V/B)) \tag{5}$$

where S is severity index and P(V/B) is probability index

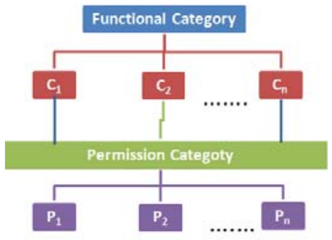

Fig. 3. Application grouping based mixed mapping matrix methodology

IV. Preliminary Results

The application types were decided by looking at their functional category and permission requests. The vulnerabilities suggested here included a variety of threats such as system crash, buffer overflow, infinite loops, memory related threat, HTTP splitting and performance. Whereas, the bugs included

Finance apps Vulnerabilities → Software Bugs ↓	User Interface Failure	Buffer Overflow	Infinite Loops	Null Point Dereference	Memory Safety Issues	Deadlock	HTTP Splitting	Cross Site Scripting	Performance Degrade
Bad Practice	Medium	High	High	High	High	High	Low	Low	Low
Correctness	Low	Low	Medium	Medium	High	High	Medium	Medium	Low
MT-Correctness	Low	Low	Low	Medium	Medium	High	Medium	Medium	Low
Performance	Low	Low	Low	Low	Low	Low	Low	Low	High
Security	Medium	Low	Medium	Medium	High	Medium	High	High	Medium
Style (Dodgy Code)	Low	Medium	Medium	Medium	Medium	Medium	Low	Low	Medium
Usability	Low	Low	Low	Low	Low	Low	Low	Low	Medium
internationalization	Low	Low	Low	Low	Low	Low	Low	Low	Low

TABLE III. BUGS TO VULNERABILITY SEVERITY MAPPING FOR APPLICATIONS WITH FINANCE GROUPING

Internet Permission Apps Vulnerabilities → Software Bugs ↓	User Interface Failure	Buffer Overflow	Infinite Loops	Null Point Dereference	Memory Safety Issues	Deadlock	HTTP Splitting	Performance Degrade
Bad Practice	0.07	0.08	0.08	0.07	0.09	0.06	0.12	0.05
Correctness	0.15	0.13	0.13	0.09	0.12	0.06	0.08	0.17
MT-Correctness	0.06	0.08	0.05	0.04	0.04	0.12	0.03	0.05
Performance	0.03	0.03	0.03	0.013	0.02	0.01	0.02	0.07
Security	0	0	0	0	0.02	0	0.03	0.02
Style (Dodgy Code)	0.08	0.1	0.12	0.1	0.1	0.05	0.08	0.1
Usability	0	0	0	0	0	0	0	0

TABLE IV. BUGS TO VULNERABILITY PROBABILITY MAPPING FOR APPLICATIONS WITH INTERNET PERMISSION

the commonly detected ones by static analysis like bad practice, security, usability, correctness, style and internationalization. From six functional and four permission categories, we show only two in Tables III and IV that depict few of the many mapping scenarios of Severity and Probability methods. The Finance application and Internet service have been exploited more than the others in their respective methods. The metric of comparison is obtained by the average summation of the ranks given by the cost function below.

$$C\_v = Severity\_v \times P(V|B) + K\_c \qquad (6)$$

where C_v is the vulnerability cost function and K_c is the static (prior) cost associated with a vulnerability

This clearly tells us how to identify the most common mobile security vulnerable apps from their bugs-to-vulnerability mapping matrices. Similar matrices were obtained for other categories of application groupings as well.

V. CONCLUSION AND FUTURE WORK

We have presented this study to systematically examine the mapping between unintentional bugs and known vulnerabilities. We examined the third party Android apps for unintentional and malicious intent, that can be triggered by downloading the app. The results are encouraging and indicate the need for further studies on this huge unexplored area of relationship between bugs and vulnerabilities, and vice-versa. We also believe that there are possibly many more vulnerabilities associated with bugs in Android based apps which were not a part of this study. In future, the obtained matrices can be improved by identifying, and adding more vulnerabilities and bugs, and determining an index for measuring the strength of a given vulnerability.

ACKNOWLEDGMENTS

This work is partially supported by the National Science Foundation under grants CNS-0751205 and CNS-1229700.

REFERENCES

[1] comScore, "Press releases reports, smartphone platform market share, dec 2014 u.s." *URL: http://www.comscore.com/Insights*, Accessed on: Jan 2015. [Online]. Available: http://www.comscore.com/Insights

[2] V. Okun, A. Delaitre, and P. E. Black, "Report on the static analysis tool exposition (sate) iv," *U.S. National Institute of Standards and Technology (NIST) Special Publication (SP)*, 2013.

[3] L. Batyuk, M. Herpich, S. A. Camtepe, K. Raddatz, A.-D. Schmidt, and S. Albayrak, "Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within android applications," in *Malicious and Unwanted Software (MALWARE), 2011 6th International Conference on*. IEEE, 2011, pp. 66–72.

[4] J. Fonseca and M. Vieira, "Mapping software faults with web security vulnerabilities," in *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*. IEEE, 2008, pp. 257–266.

[5] D. Mitropoulos, V. Karakoidas, P. Louridas, G. Gousios, and D. Spinellis, "Dismal code: Studying the evolution of security bugs," in *Proceedings of the LASER Workshop*, 2013, pp. 37–48.

[6] J. Wu, Y. Wu, M. Yang, Z. Wu, and Y. Wang, "Vulnerability detection of android system in fuzzing cloud," in *Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing*. IEEE Computer Society, 2013, pp. 954–955.

[7] W. Jimenez, A. Mammar, and A. Cavalli, "Software vulnerabilities, prevention and detection methods: A review1," in *Proc. European Workshop on Security in Model Driven Architecture*. Citeseer, 2009, pp. 6–13.

[8] G. Schryen, "Is open source security a myth? what do vulnerability and patch data say?" *Communications of the ACM (CACM)*, vol. 54, no. 5, pp. 130–139, 2011.

[9] Google, "Android tools project site - android lint," *URL: http://tools.android.com/tips/lint*, Accessed on: Feb 2013. [Online]. Available: http://tools.android.com/tips/lint

[10] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, ser. PASTE '07. New York, NY, USA: ACM, 2007, pp. 1–8. [Online]. Available: http://doi.acm.org/10.1145/1251535.1251536

[11] D. Hovemeyer and W. Pugh, "Finding more null pointer bugs, but not too many," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, ser. PASTE '07. New York, NY, USA: ACM, 2007, pp. 9–14. [Online]. Available: http://doi.acm.org/10.1145/1251535.1251537

[12] S. Christey, "Unforgivable vulnerabilities," *The MITRE Corpotarion*, 2007.

[13] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.