Attack Surface Measurement on Android Applications

Kevin Campusano
B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York 14623
Email: kac2375@rit.edu
Advisor: Dr. Andy Meneely

I. Introduction

Mobile devices have seen an explosive increment in use during the past few years. With the advent of tablets, phones and other smart devices, computing has taken a new, more accessible and portable form and has become integral to our everyday lives now more than ever. With processing power comparable to conventional desktop machines, these devices are capable of meeting our most common computing needs. To achieve this, smart devices access and manipulate sensitive information such as location, calendars and emails. Indeed, smart devices provide a lot of convenience. However, due to the amount of sensitive information they manage, they also represent a security risk.

In this day and age only a handful of different mobile platforms hold the majority of the market share. Among those, Android has seen an outstanding growth in its install base and now holds a privileged position of more than half of smartphone users in the US [1]. Due in part to its popularity, the Android platform has been the target of multiple attacks [2]. Even though Android's application development framework and the operating system itself provide (although not perfect [3]) many defense mechanisms [4] [5], the responsibility to protect sensitive information ultimately falls to application developers as their applications are what interact directly with the users and their information.

With security being such an important concern, the need for it to be managed is imperative. For software security to be managed and subsequently improved, measuring it, as is the case with most quality attributes, is essential. To address this problem we propose the use of a measurement of applications' attack surface based on their call graph information [6].

The attack surface is defined as all the different ways in which a malicious user can take advantage of a system and compromise its data. It can be thought as the attackability of a system. We can say then, that the smaller a system's attack surface is, the more secure it is [6]. It makes sense then that a metric for the attack surface would be relative. As such, we propose measuring the attack surface of different versions of an application and comparing the various measurements to discover its evolution and whether it has become more or less secure.

To guide our research efforts, we propose the following research questions:

Can the attack surface of an Android application be measured? How does the attack surface evolve in Android applications?

Past research in the area of Android application security has focused on enhancements to the application development framework's security aspects [7], application privilege [8] and inter-application communication [9]. There has been previous work on attack surface of android [10] that used call graph information of applications to identify permission gaps. Our approach proposes the use of call graph information to measure and characterize the attack surface and use this to observe the difference in security between various versions of studied applications.

II. Android Application Development Platform Overview

The Android platform provides developers with a rich application development framework that contains a varied collection of libraries for most common functionalities and for interaction with the device's physical components. This development framework provides a core set of building blocks that most Android applications are constructed with:

A. Activities

The most common of the framework's main building blocks. All applications that involve some sort of interaction with the user are composed of at least one Activity. Activities are what provide user interfaces for the applications. In a traditional sense, each activity would correspond to a screen, so to speak. They contain both graphical user interface definitions and code.

B. Services

Services embody operations that run in the background without any interaction with the user. Downloading a file or continuously keeping email synchronized are some examples of functionalities that are typically implemented as services.

C. Broadcast Receivers

These are components that are responsible of receiving messages (i.e. Intents) from other applications or services that are intended to be handled for multiple applications. More often than not, a Broadcast Receiver's sole purpose is to relay the received messages to an Activity or Service that supports the particular operation specified in the Intent.

D. Content Providers

These are shared storage resources that can be accessed by multiple applications using their assigned URIs. These are used for both data persistence and information sharing between apps.

E. Intents

Android's inter process communication mechanisms are built around Intents. These are objects that encapsulate a message which typically contains a recipient, an operation and some data to work with. Intents can be sent both from the operating system and applications and can be handled by Activities, Services and Broadcast Receivers.

Depending on how the recipient for the operation described in the intent is specified, these can be of two types: Explicit and Implicit. Explicit Intents specify their recipient application while the Implicit ones describe their recipient as any application that meets certain criteria (i.e. that support the operation that the Intent represents). Implicit Intents rely on the Android platform to find a suitable application to handle it. The operating system also allows the user to select which application to handle such operations if multiple applications that support that functionality are installed in the device.

III. Android Security Model Overview

Android is in essence a linux based operating system build from the ground up with a focus on mobile devices and security as one of its main design goals. As such, there are security mechanisms put in place at the operating system level that affect how the users interact with their devices.

A. The Market

There are many different virtual markets from where users can obtain Android applications. The biggest one of these is the Google Play store. One of the core objectives of the Android platform is to keep it accessible both to developers and users. The process of getting applications in the store is simple and the cost is minimal. While this strategy is sound for ensuring the growth and relevance of the platform, it is not as sound for security purposes. Ill intentioned developers won't find much resistance in getting malicious applications into the Android market which will subsequently reach consumer devices and potentially compromise them and their users' data.

B. Permissions

Android's security model is permission based. This means that the concept of permissions plays an integral role in the security of the platform. These permissions are what control the applications' access to the user's personal data store in the device as well as access to any physical component on the device. Android permissions range from access to the user's contacts information to the ability to use of the camera installed in the device.

Prior to installing an application, the underlying operating system prompts the user with the permissions that the application about to be installed requires to function. The user has the option to accept or decline installing the application after (hopefully) carefully reviewing the permissions requested. These permissions are specified by the developers as part of the application's source files.

C. Runtime

At runtime, all the applications that are installed in the device are executed in isolation. The kernel provides a sandboxing mechanism that allows the applications to run separate to each other and to the rest of the system functions. In their sandbox, each application has its own set of resources that are not shared with any other running process. Each application runs in a different instance of the interpreter.

This design prevents any application from accessing private data from the system and from other applications.

D. Inter-application Communication

Even though, by design, applications run in isolation in the Android platform, there are mechanisms for interprocess communication in place. In Android, interprocess communication is achieved by one of the basic building blocks of the Android application development framework: Intents. Intents are objects that can be instantiated by any application and the operating system itself and encapsulate a message that can be sent to other running processes. This message generally contains information for the receiver about what operation to perform, what data to work with or both.

When Intents are sent, they can be directed to a specific application or to any application that supports handling the particular action specified in the Intent. For example, an application that requires the camera may not directly use it but invoke the default camera application installed in the device. For this, the requesting application will create an Intent and configure it to be handled by the camera application. Likewise, the operating system could use Intents to notify a power saving application when the battery life has reached a certain threshold of consumption.

IV. Related Work

There has been extensive work in the area of security in the Android platform.

A. Android Security

Shabtai et al. [3] performed a comprehensive security assessment of the Android platform. They evaluated all security concepts and mechanisms used in the platform. Both those that are inherited from the Linux kernel and new to the Android operating system itself. They identified potential weaknesses on the security mechanisms of the platform and offered several techniques that could be used to address these weaknesses.

Gommerstadt et al. [11] developed a model of the information flow in Android applications with a focus on the flow of private and sensitive data. They use previous studies on Android security as a base for their model and present two applications as case studies as a way for using their model to study the flow of information.

B. Taxonomy of Android Attacks

Vidas et al. [2] from the Carnegie Mellon University developed a taxonomy of all the known attacks that targeted the Android platform. The classification they proposed grouped the attack classes into two categories depending on the access that the attacker would need on the device: Attacks with physical access and attacks with no physical access.

1) Attacks without physical access: For these types of attacks, the attacker need not have physical access to the targeted device. Generally, these attacks consist on the attacker finding the way to execute malicious code on the device. In essence, this means that the attacker convinces the user to trust and run their code one way or the other. Once the malicious code is running on the targeted device, any vulnerability can be exploited to obtain privileged access and compromise the user's sensitive data or cause other manners of harm.

Unprivileged Attacks: Although the way that most attacks can maximize the damage they cause in a device is by obtaining elevated privileges, there is still fair amount of harm that can be done without breaking free from the Android security model. Still within their sandbox, with standard application permissions, malware can be dangerous.

Some of the ways that seemingly benign applications can reach a user's device is by installing them directly from the Internet bypassing and ignoring any operating system warning as well as installing applications that request a large number of permissions at install time with dubious purposes. The Google Play Store as well as many other Android markets provide a web based interface that users can use to remotely install applications to their devices. If an attacker were to somehow hijack the user's session or authentication credentials, then they would have the ability to install any malicious application to the user's device and potentially compromise it. In addition to these methods, application repacking is a threat present in Android markets. Application repacking consists on the attacker downloading and decompiling existing popular applications, injecting malicious code into them and reuploading them to the market. Since these repacked applications are identical to their legitimate counterparts, users are tricked into installing them without a second thought and subsequently running malicious code on their device.

Privileged Attacks: Attacks consisting on obtaining elevated privileges and remotely executing code are based on many of the same techniques discussed earlier. The attacker somehow finds a way to get malicious code to execute on the targeted device, the difference is that this code's purpose is to perform a privilege escalation attack. Code that executes with elevated privilege in the device is outside the restrictions of all of the platform's security mechanisms and as such has a great potential of causing harm.

These kinds of remote exploitation attacks can also be achieved without the installation of malicious software in the target device. Vulnerabilities in common software that mobile devices use like web browsers or even the underlying Linux kernel can be exploited to run harmful code with high privilege.

Bugiel et al [8] explore the problem of developing a framework to protect the Android platform against two specific types of privilege escalation attacks: confused deputy and collusion attacks. They contribute with ways to improve the security of Android against these types of attacks.

2) Attacks with physical access: Of course, physical security is still a valid concern. This category describes the attacks that can be done when the attacker obtains physical access to the device.

ADB enabled: The Android Developer Bridge is a program that aids in the development and debugging of Android applications from a computer connected to a device. If an attacker gains physical access to a device where the Android Developer Bridge is enabled they can hook it up to a computer that has the Android developer tools installed and interact with the device. This can be done even if the device is obstructed with mechanisms such as lock screens. Via the ADB, the attacker is able to run untrusted code in the device and, as a consequence, compromise user information. Privilege escalation attacks can also be performed this same way by running the exploits from the ADB.

ADB disabled: If an attacker gains access to a device that is obstructed and in which the ADB is disabled, there are still options available to take advantage of it. The way to attack such a device is by booting the device in recovery mode using a forged recovery image. When booting from this image, the attacker has total control over the device and can run malicious code or gain elevated privileges.

On unobstructed devices: This is the best case scenario for the attacker. If the attacker gains physical access to an unattended device that is not obstructed in any way (e.g. no screen lock) then the effort needed to run malicious code in it is minimal. In this situation any of the previously discussed attack methods are available.

Based on their proposed taxonomy and the attacker's capabilities, Vidas et al. also designed a flowchart of how an attacker would approach a breach into an Android system. It can be seen in Figure 1.

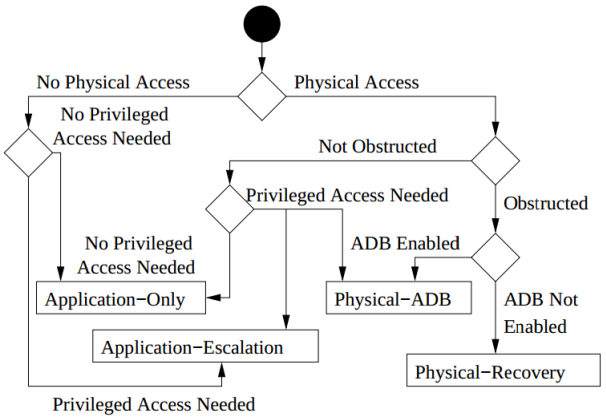

Fig. 1. Android attack flowchart developed by Vidas et al.

C. The Attack Surface

As Manadhata and Wing [6] put it: "A system's attack surface is the subset of its resources that an attacker can use to attack the system." In their work Manadhata et al. [6] come up with a definition for a multi-dimensional metric for attack surfaces based on many factors. One particularly important is the count of Entry and Exit Points of the system under analysis. They develop a methodology for obtaining systems' Entry and Exit Points based on their call graph. Both static (compile-time) and dynamic (runtime) data is needed for a comprehensive call graph.

In subsequent work, Manadhata et al. [12] [13] use their developed methodology to measure and study the attack surface of industrial and open source software. In [12] they present a case study on two applications in the same domain. The authors apply the concepts explored in their previous work and put them to practice with two case studies. They use their multi-dimensional attack surface metric to compare the security attribute of two Unix FTP Daemons: ProFTPD and WuFTPD. This paper provides a real-world scenario in which an attack surface metric could be used to support software acquisition decisions by comparing the attack surface of two or more software alternatives.

In [13] they present another case study, this time with an enterprise software written in Java: SAP. They implement their metric calculation methodology as an eclipse IDE plugin. In this case study, they present their metric as a complementary approach to general code quality assessments for software security.

D. The Attack Surface in Android

Previous work on attack surface of Android applications has focused on inter-application communication [9] and permission gaps [10]. Bartel et al. [10] focus on the permission-gap and uses that to define the attack surface in Android applications. They define this as the mismatch between the permissions that an application requests at installation-time and the ones that it actually uses. Their premise is that the bigger the difference between permissions quantity the larger the attack surface because there are more permissions requested but not needed. They develop a static analysis based method to extracting the call graph information of Android applications calculate their permission gap.

Chin et al. [9] explore another aspect of Android applications' security related to the attack surface: inter-application communication. They explain the risks that come with inter-application communication and present a tool they developed to analyze applications and detect vulnerabilities.

Our approach proposes the use of call graph information to measure and characterize the attack surface and use this to observe the difference in security between various versions of applications.

V. Methodology

In our study, we plan to develop a method for measuring the attack surface of Android applications based on Entry Points and Exit Points obtained from their call graph information. By definition, Entry Points are the functions, methods or procedures through which data flows into the system. Similarly, Exit points are those through which data flows out of the system. Simply put, Entry Points are methods that contain calls to APIs for data input (i.e. Input Methods) and Exit Points are methods that contain calls to APIs for data output (i.e. Output Methods). The basic process for measuring the attack surface in terms of Entry Points and Exit Points involves obtaining the applications call graph from the source code and identifying all the Entry and Exit points in the call graph.

The initial efforts in this project will be geared towards building knowledge about the Android platform that will allow us to obtain the call graph and identify the Entry and Exit Points. The first step of our study will be to survey the Android application development framework and identify all the Input and Output Methods present in the libraries. These Input and Output Methods will serve as a basis for identifying the Entry and Exit Points once the call graph is obtained. As a result of this step, we expect to have a list of all the Input and Output Methods that can be programmatically used to identify Entry and Exit Points given a particular call graph.

During this step we will also look for tools that will allow us to obtain the call graph of Android applications. Given prior experience, we expect these tools to rely on either source code static analysis or runtime monitoring of executing applications. A combination of the two approaches may be necessary to obtain a comprehensive call graph. In our approach, the soundness of the call graph is essential to the measurement of the attack surface.

After this, the next step would be to use this knowledge to implement an automated tool that will allow us to, given an application's source code, invoke the call graph generation utilities discussed earlier and parse whatever output format we obtain from them and turn it into an in-memory data structure that can be operated on to obtain a diverse set of metrics based on graph analysis.

When the tool has been successfully developed and tested, we will proceed to obtain the source code of a variety of

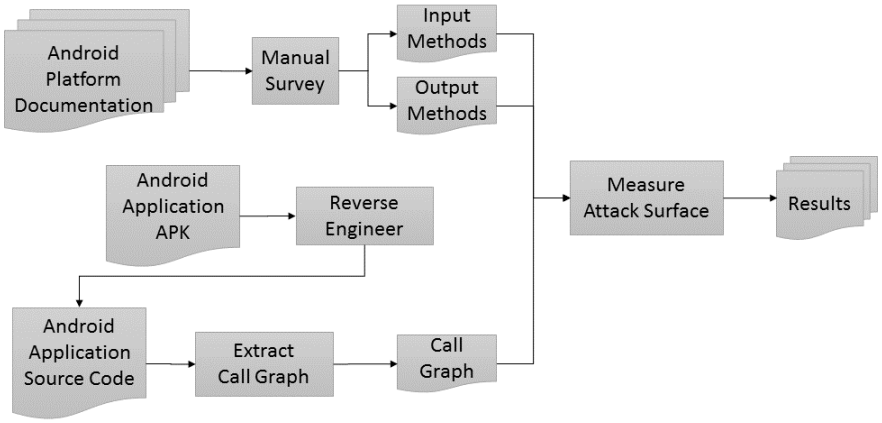

Fig. 2. Overview of the attack Surface measurement methodology

Android applications and measure their attack surface. Figure 2 shows an overview of the proposed methodology for attack surface measurement. We plan to divide this stage of the study into two phases: with the majority of the applications we will perform a single measurement and on a select few we will delve in more deeply and perform an evolution analysis. some of the applications we select as case studies will be analyzed with more depth by observing the change attack surface across releases. For this, we will obtain the source code of said applications as they were at different points in time and make our measurements on these code bases.

To maximize our sample, we will obtain the case study applications' source code by downloading the installation packages (i.e. APKs) from the Google Play Store (or any other Android store) and decompiling them. This way we won't be limited by the availability of open source applications. For our evolutionary analysis, however, we will still need access to the source code repositories of the applications we select to perform this study on. Hence, we will be limited to open source applications for this more detailed study.

VI. Tools

Python: The Python [14] programming language will be used for developing the scripts and tools needed for measuring the attack surface. In previous work, we have developed a tool that, given an in-memory representation of a call graph, calculates various metrics. We plan to extend this tool with support for Android applications. For this we will need to develop an Android call graph parser component that will take a text representation of the call graph on an application and convert it to an in-memory data structure that can be operated on. Other enhancements will need to be done in order to support the new Entry and Exit Points definitions that will be derived from studying the Android application development framework.

Networkx: Networkx [15] is a Python library that offers graph analysis functionalities. Once the call graph is translated into a data structure that Networkx can work with, it will be used to calculate various types of metrics.

Git: The applications that we select for performing the evolutionary analysis will most likely have their source code version controlled in Git [16]. We will use the many features provided by the Git command line interface to mine the repositories and obtain multiple versions of the applications we select as case studies for the evolutionary analysis.

android-apktool: This tool [17] will allow us to reverse engineer the Android application packages (APKs) containing the applications that we select as case studies and obtain their source code for analysis.

java-callgraph: We will use this tool [18] to obtain the call graph information of the android applications we select as case studies.

Spark: An alternative call graph generation tool for java that [19] is part of the Soot analysis framework [20]

VII. Work Schedule

Figure 3 shows the plan of work with all the project's activities with assigned beginning and finishing dates.

References

[1] Whitney, Lance. "Android Loses Some US Market Share but Remains Top Dog." CNET. N.p., 4 Sept. 2014. Web. 19 Oct. 2014.

[2] Vidas, T., Votipka, D., & Christin, N. (2011, August). All Your Droid Are Belong to Us: A Survey of Current Android Attacks. In WOOT (pp. 81-90).

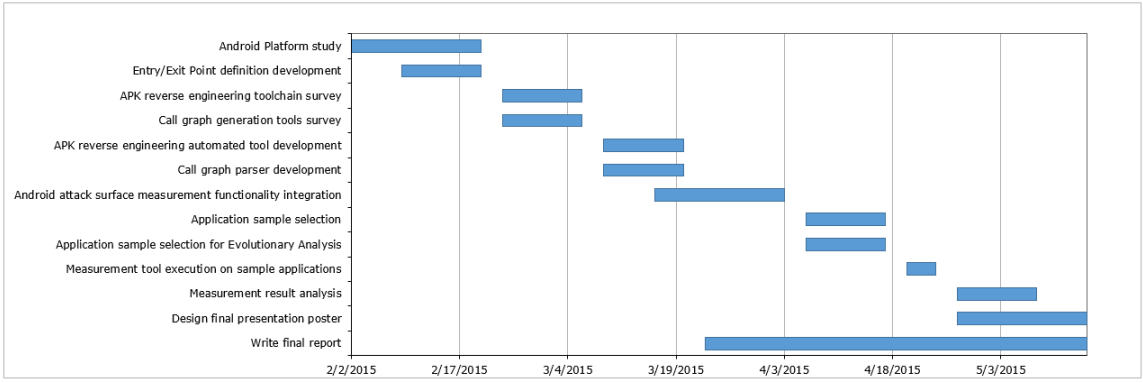

Fig. 3. Project work schedule

[3] Shabtai, A., Fledel, Y., Kanonov, U., Elovici, Y., Dolev, S., & Glezer, C. (2010). Google android: A comprehensive security assessment. IEEE Security and Privacy, 8(2), 35-44.

[4] Burns, J. (2009). Mobile application security on Android. Black Hat, 9. Enck, W., Octeau, D., McDaniel, P., & Chaudhuri, S. (2011, August). A Study of Android Application Security. In USENIX security symposium (Vol. 2, p. 2).

[5] Manadhata, P. K., & Wing, J. M. (2011). An attack surface metric. Software Engineering, IEEE Transactions on, 37(3), 371-386.

[6] Ongtang, M., McLaughlin, S., Enck, W., & McDaniel, P. (2012). Semantically rich applicationcentric security in Android. Security and Communication Networks, 5(6), 658-673.

[7] Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A. R., & Shastry, B. (2012, February). Towards Taming Privilege-Escalation Attacks on Android. InNDSS.

[8] Chin, E., Felt, A. P., Greenwood, K., & Wagner, D. (2011, June). Analyzing inter-application communication in Android. In Proceedings of the 9th international conference on Mobile systems, applications, and services (pp. 239-252). ACM.

[9] Bartel, A., Klein, J., Le Traon, Y., & Monperrus, M. (2012, September). Automatically securing permission-based software by reducing the attack surface: an application to android. In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (pp. 274-277). ACM.

[10] Gommerstadt, H., & Long, D. (2012). Android Application Security.

[11] Manadhata, P., Wing, J., Flynn, M., & McQueen, M. (2006, October). Measuring the attack surfaces of two FTP daemons. In Proceedings of the 2nd ACM workshop on Quality of protection (pp. 3-10). ACM.

[12] Manadhata, P. K., Karabulut, Y., & Wing, J. M. (2009). Report: Measuring the attack surfaces of enterprise software. In Engineering Secure Software and Systems (pp. 91-100). Springer Berlin Heidelberg.

[13] Python (n.d.). Retrieved November 24, 2014, from https://www.python.org/

[14] Networkx (n.d.). Retrieved November 24, 2014, from https://networkx.github.io/

[15] Git (n.d.). Retrieved November 24, 2014, from http://git-scm.com/

[16] android-apktool (n.d.). Retrieved November 24, 2014, from https://code.google.com/p/android-apktool/

[17] java-callgraph (n.d.). Retrieved November 24, 2014, from https://github.com/gousiosg/java-callgraph

[18] Soot Analysis Framework (n.d.). Retrieved November 24, 2014, from http://www.sable.mcgill.ca/soot/doc/overview-summary.html

[19] Lam, P., Bodden, E., Hendren, L., & Darmstadt, T. U. (n.d.). The Soot framework for Java program analysis: a retrospective. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.221.5311