CRAB-Droid

AJ Arnold, Matthew Berthoud, Justin Cresent, Ada Rigsby

May 11, 2024

Abstract

Analyzing Android applications for correct security practices such as using permissions, using the SSL API, and potential interface vulnerabilities is a challenging task. This paper presents our tool CRAB-Droid, a python script that builds upon Androguard to carry out a static analysis to identify potential security vulnerabilities in Android applications. We analyzed 100 apps from the Google Play Store with the objective of identifying potential security vulnerabilities. We created 5 experiments to test on our 7 research questions we had outlined in the planning stage of our analysis. Our findings ... #TODO This project shows the significance of developers following best practices when creating Android apps, and the abilities to develop tools to help guide and educate engineers to them. Implementing tools like Androguard and CRAB-Droid to be used in the development process can help to identify potential security vulnerabilities before putting apps into deployment onto the app stores. This can help to save time, money, and most of all, user privacy.

1 Introduction

Currently, Android is the most used smartphone operating system in the world, with a market share of 48% and over 400,000 applications (apps) available in the Google Play Market [4]. The Google Play Market is also mostly open and unrestricted, allotting developers more freedom, but many times at the cost of security. The ability to identify vulnerablities within Android applications is vital to the safety of users and their information. Showcasing these vulnerabilities then helps to educate developers to carry-out best security practices when creating an app and to take vulnerable apps off the app store to be secured. This project investigates a variety of Android applications and vulnerabilities that potentially exist within them.

With technolgocial enhancements to smartphone devices over the past two decades, user have been downloading more and more apps to their phones. This rise of phone apps results in a greater chance for users to download malicious ones that may contain malware, such as trojans [3]. This spike has also lead to many new developers to enter the ever expanding market, with many potential bad security practices in apps along with them. For reasons mentioned previously, it is important to identify these vulnerabilities in order to educate developers and protect users.

For this project, we analyzed a set of 100 Android apps from the Google Play Store. We then established 7 research questions to hypothesize about pontential vulnerabilites we believed we would find in the apps. We organized them into three domains: permission misuse, SSL API misue, and interface vulnerabilites. These domains were chosen to cover a broad attack vector of the android applications such as the use of permissions that may violate user privacy, the use of SSL APIs that may not be secure, and the use of interfaces that may be vulnerable to attacks. Five experiments were then developed and tested on our set of apps.

The five experiments were tested on each app as follows: test apps for whether they use all the permissions they request and if they use any dangerous combinations of them, examine for an overide on trust mangers or error handlers, test for implementation of the AllowAllHostnameVerifier class, see if apps contain mixed use of HTTP and HTTPS (mixed SSL), and finally test apps for incorrect use of the addJavascriptInterface method.

Using the Androguard library [1] and referencing Mallodroid [4], we completed a thorough analysis of the apps. The results showed ...

Overall, the significance of this project is seen in its use of static analysis and its applications into the discipline of Android mobile app security. This practice has the potential to be utilized in real-world applications to help secure the Google Play Store and protect users from malicious apps.

2 Methodology

This section presents the methodology for how this project sets out to take a set of apps to analyze, create automated experiments to evaluate each app, and then generate results to view true and false vulnerabilites detected within the apps.

2.1 Set of Apps

2.2 Experiment Design

2.3 Results Generation

3 Experiments

3.1 Permissions Misuse

Many android applications misuse permissions that they allow their application to possess. Often, certain permissions are granted to an app that doesn't use them in the first place. This practice is dangerous: it can leave vulnerabilities within the application that should not be possible based on the actual functionality of the application. For example, it would be easy for a developer to add functionality to an app that requires a certain permission, and then leave the permission in after reverting their previous changes.

Another common theme seen within applications is the combination of two permissions that can create a dangerous comibination. For example, a malicious application with the permission combination of CAMERA and RECORD_AUDIO would allow the application to have access to a devices camera and microphone, tools that would be able to perform serious invasions of privacy. Another example could be seen with INTERNET and ACCESS_FINE_LOCATION, which would grant an application the ability to track a device's physical location (and subsequently serve as a tool for stalking).

The ability of permissions to allow an application within reach of sensitive data means that users should be prompted whether they wish to allow certain permissions to be enabled. However, many applications do not give users the chance to make this decision; this choice is a significant breach of trust between an application and its users.

All of these developer mistakes constitute permission vulnerabilities within android applications. For our first experiment, we will be testing these common misuses of permissions. First, the experiment will test whether apps utilize all of the permissions that they request. Second, the experiment will search for the use of dangerous permission combinations implemented in the app. Finally, the experiment will test whether users are prompted to explicity give their consent to every permission used.

Setup:

3.2 Trust Managers and Error Handlers

Trust Managers are put in place to verify the authenticity of a remote server. To do this, many android built-in trust managers are implemented to securely verify a server's certificate. However, the built-in X509TrustManager class allows the complete override of the server verification process, potentially endangering an application if implemented incorrectly.

Many times, developers will avoid the built-in trust manager in an effort to take shortcuts around the correct implementation (whether this be for convenience or lack of experience). This practice is often carried out by implementing the checkServerTrusted() and getAcceptedIssuers() functions in a way that configures the hostname verifier to trust all X.509 certificates. By doing this, developers expose their application to danger; third parties may attempt a Man-in-the-Middle attack on network traffic from the application, compromising a user's network data if successful.

TALK ABOUT ERROR HANDLING HERE

This experiment will be testing whether or not an app overrides a built-in trust manager or error handler to forgo either method's intended purpose of correctly verifying certificates.

Setup:

3.3 AllowAllHostnameVerifier

The HostnameVerifier interface within Android Studio is responsible for the verification of the hostname within the server being connected to, making sure the hostname within the server's certificate matches the one seen in the server the client is attempting to connect to.

A vulnerability arises when the developer attempts to shortcut the hostname verification process (similar to Experiment 2), resulting in an ineffective verification process. Specifically, many developers use

3.4 Mixed use SSL

3.5 addJavascriptInterface Method

4 Evaluation

This section comprehensively evaluates the performance of our proposed and tested experiments. It determines which experiements performed the best on criteria of most true positive matches and least false positive matches.

5 Results

6 Findings and Future Work

6.1 Analysis of Results

6.2 Future Work

7 Conclusion

References

[1] Sebastian Bachmann Anthony Desnos, Geoffroy Gueguen. Androguard. https://github.com/androguard/androguard, 2018.

[2] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, page 239–252, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306430. doi: 10.1145/1999995.2000018. URL https://doi.org/10.1145/1999995.2000018.

[3] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, page 235–245, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605588940. doi: 10.1145/1653662.1653691. URL https://doi.org/10.1145/1653662.1653691.

[4] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why eve and mallory love android: an analysis of android ssl (in)security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 50–61, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450316514. doi: 10.1145/2382196.2382205. URL https://doi.org/10.1145/2382196.2382205.

[4] [1] [2] [3]