CRAB-Droid

AJ Arnold, Matthew Berthoud, Justin Cresent, Ada Rigsby

May 12, 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 [**?**]. 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 [**?**]. 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 [**?**] and referencing Mallodroid [**?**], 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 outlines the methodology we will use for our experiments. We will use a set of ninety-seven applications during testing, using automated experiments to analyze each and every one. Finally, the results of these experiments will be analyzed for any true and/or false vulnerabilites present within each app.

2.1 Set of Apps

We have been given a set of a hundred .apk applications to test (we tested on 97), many of which contain different purposes, uses, features, and developers. There are applications ranging from "Live Earth Map" to "Universal TV Remote Control". This variety and quantity of applications gives us a broad scope to test and is more representative of the population of all applications you can find on Google Play.

2.2 Experiment Design

Our experiments use lightweight static analysis to parse through each application and find vulnerabilites. We utilize Androguard libraries within our CRAB-droid script to help with the decompilation and searching of the 97 applications.

Androguard is a python-based tool used for the reverse engineering of Android applications. It takes raw Android Packages (.apk) files and breaks them down, making them easier to analyze. The capabilities of the library make it a great tool for testing the existence of vulnerabilites within applications.

After our script finished, we then manually reviewed our results to identify trends with the applications.

2.3 Results Generation

Our initial script (that scrapes the Android Packages) outputs results into a text file, which is subsequently analyzed by another script. The purpose of the second script is to collect and organize our findings by displaying different trends and statistics.

Often, false positives can be found within our tests; our script identifies potential vulnerable patterns, but it does not guarentee that each individual finding is a true positive. In this way, our script generates many false positives.

3 Experiments

3.1 Permissions Misuse

Many Android applications misuse permissions that they allow their applications 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:

For this experiment, we first looked into whether the app abided by the principle of least privilege. Out of all permissions that the app asked for, we checked how many were used and how many were unused. When permissions were found to be requested, but never used throughout the app, this signified that the app did not follow the principle of least privilege, as unnecessary permissions were included in the app.

We also checked to make sure that all permissions that were included in the app were requested, providing transparency about the permissions that the app utilized.

Finally, we checked for dangerous permission combinations in the applications. We specificially looked for combinations of RECORD_AUDIO and INTERNET, ACCESS_FINE_LOCATION and RECEIVE_BOOT_COMPLETED, CAMERA and INTERNET, as well as SEND_SMS and WRITE_SMS.

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() function 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 methods intended purpose of correctly verifying certificates.

Setup:

For this experiment, we utilized Androguard and aspects of Mallodroid to check if built-in trust manager or error handler methods, specifically checkServerTrusted() and onReceivedSslError(), were overriden.

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 the AllowAllHostnameVerifier class; this class essentially turns hostname verification off (by allowing all hostnames) and instructs the process to never throw an SSL Exception.

This vulnerability, similar to avoiding trust managers, creates the opportunity for a Man-in-the-Middle attack. If the host cannot be verified, a third-party has the opportunity to impersonate a legitimate server and trick the application into sending sensitive data to it.

This experiment will test whether or not an application implements the AllowAllHostnameVerifier class, which allows all hostnames to be accepted for a certificate.

Setup:

For this experiment, we made use of Androguard to analyze an app's method instructions, checking if the AllowAllHostnameVerifier class was ever implemented. Specifically we looked for AllowAllHostnameVerifier and SSLSocketFactory - ALLOW_ALL_HOSTNAME_VERIFIER, similar to Mallodroid.

3.4 Mixed use SSL

When an application is connected to the internet, it is not good practice to used both HTTPS and HTTP. HTTPS connections are more secure since they use Secure Sockets Layer (SSL) to encrypt normal HTTP requests and responses, which consist of only plaintext messages. When HTTP content is loaded by an HTTPS page, attackers have the opportunity to read and/or modify HTTP traffic. This results in a mixed-use SSL vulnerability.

Developers create this vulnerability when they do not ensure that every resource on their page is loaded over HTTPS, and this can prove tricky; modern websites often load several different resources from various places, making it hard to keep track of where HTTPS and HTTP is used. Many times, the solution (for a developer) is to simply place an "s" within "http://" to apply SSl to it. The consequence of this vulnerability is the potential for an attacker to perform a Man-in-the-Middle attack.

In this experiment, we will test whether or not an application is using a mixture of HTTP and HTTPS protocal when loading content.

Setup:

For this experiment, we will parse through class files to locate instances where the developer is using certain libraries (such as "HttpURLConnection"), and whether or not said libraries are configured to use HTTPS consistently. This will involve string searches for "http://" and "https://" to see whether an app is only using HTTP (or if it is mixed use).

3.5 addJavascriptInterface Method

Many applications use WebViews as a way to display web pages as a part of their activity layout. The addJavascriptInterface method is subsequently used to inject a supplied Java object into a WebView. This process allows JavaScript to control the host application, but presents a significant security threat if a developer is not using the interface only where necessary.

The vulnerability arises when a malicious 3rd-party seeks to use the webview as a bridge into the users system. If a webview were to contain untrusted content, an attacker could use it to manipulate the host application in unintended ways. This is done by injecting JavaScript into a user's system and running the malicious code.

By using the @JavascriptInterface annotation, developers can use the interface only where necessary and avoid allowing untrusted content within webviews.

This experiment will test whether or not the application is being exposed to potentially dangerous outside sources, depending on if the application is using a webview.

Setup

For this experiment, we will use keyword searches to

locate instances of the addJavascriptInterface() method. If it is successfully found, we will subsequently search for the use of the @JavascriptInterface annotation, confirming whether or not the developer is using the method responsibly.

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

1. Experiment 1: Permission Misuse

   90 of the 97 apps contained permissions that were unused (and thus unnecessarily added). The average number of unused permissions used within all apps was 6.01 , and when only considering apps with at least one unused permission, this number rises to 6.48.

   62 of the 97 apps were identified to be using a dangerous combination of permissions. The average number of dangerous permission combinations within all apps totaled 1.21, with the number rising to 1.89 when only considering apps with at least one instance of a dangerous permission combination.

   None of the apps requested no permissions.

2. Experiment 2: Trust Managers and Error Handlers

   XX of the 97 apps contained overriden trust managers. The average number of overridden trust managers within the apps totaled XXX, with the number rising to XXX when only considering apps with at least one instance of an overridden trust manager.

   XX of the 97 apps contained overridden error handlers. The average number of overridden error handlers within the apps totaled XXX, with the number rising when only considering apps with at least one instance of an overriden error handler.

3. Experiment 3: AllowAllHostnameVerifier

   20 of the 97 apps contained the AllowAllHostnameVerifier class. The average number of AllowAllHostnameVerifier uses within all apps totaled to .25, with the number rising to 1.26 when only considering apps with one instance of the class.

4. Experiment 4: Mixed use SSL

   90 of the 97 apps contained mixed use SSL. None of the apps contained only HTTPS or HTTP usage, and 5 of the apps used no URLs at all. The average number of HTTP URLs within the apps totaled to 49.51, with this number rising to 52.77 when only considering apps with at least one HTTP URL being used.

5. Experiment 5: addJavascriptInterface Method

   80 of the 97 apps contained improper handling of the addJavascriptInterface method. The average number of javascript vulnerabilites totaled to 3.67 per app, with the number rising to 4.75 when only considering apps with at least one vulnerability of this kind.

6 Findings and Future Work

6.1 Analysis of Results

6.2 Future Work

If we were to continue this research in the future, we would want to explore some of the following areas.

First and foremost, we would like to have more though experiments. Our experiments currently utilize the Androguard library in order to parse the apk files and find simple string matches. One example is from experiment two, where we search for overridden built-in methods. This experiment is fairly simplistic and could be expanded upon into something where the method internals are also automatically checked to view if they are forgoing original intended use, making the user vulnerable.

Additionally, we would like to expand our set of Android applications to test on. If we were able to test on a larger set of apps, this would allow us to draw more accurate conclusions and notice more common trends in regards to vulnerabilities. This would allow us to develop a more comprehensive tool and experiments.

Finally, instead of just utilizing static analysis, breaking into the domain of dynamic analysis could prove to be beneficial in seeing how these vulnerabilities can be exploited in real time. This would give us a chance to see how our predictions of vulnerabilities holds in a test environment of the application running. The results could prove valuable to enforce our findings and potentially find more vulnerabilities that were not found in static analysis.

7 Conclusion

This project explored developing and evaluating experiments that sought out to find potential vulnerabilities within a set of Android apps. After downloading the apps, we then developed a set of five experiments to cover three

domains: permission misuse, SSL API misue, and interface vulnerabilites.

Our work shows the importance of understanding the permissions that an app requests along with their implications to be used or used together potentially maliciously by attackers. Another factor is the importance of understanding the SSL API calls that an app makes. Communicating securely over the internet with the use of SSL is vital to the privacy and security of users data and personal information. Our tool developed in this project showcases the implications of scanning Android applications for potential vulnerabilities before entering or while in the market, or to catch in the early stages of development. Tools like this could help aid developers and security professionals alike.

Main takeaways include:

1. Static analysis is a powerful baseline in order to target specific sectors within an app in order to identify potential vulnerabilities and attack vectors.

2. This project's result signifies the importance of best security practices when developing and publishing Android applications.

3. The graphs and tables help to visulize the data and results to see trends and potentially repeated bad practices and vulnerabilities.

Overall, this project serves as a stepping stone for an ever growing field of work. With mobile phone technology improving each year, the need for secure applications is more important than ever. Tools like CRAB-Droid can quickly and effectively aid developers and security professionals to identify potential vulnerabilities within applications and minimize the threats awaiting their users.

[**?**] [**?**] [**?**] [**?**]