Analysis of Android Applications

Andrew Lam (32782120), Caleb Kwan (92859115), James Tu (10798122), and Saurabh Vishwakarma (55898126)
Department of Computer and Electrical Engineering
University of British Columbia
Vancouver, British Columbia
andrew8b2@gmail.com, caleb.kl.kwan@gmail.com, jat2872@gmail.com, svishwak4@gmail.com

Abstract—**The Androlyzer is a tool that uses static analysis to detect security flaws as well as dependency issues in Android applications. We conducted surveys to compare this tool with other tools which were currently available on the market, and found that our tool was significantly easier to use and understand.**

I. Introduction

The complexity of analyzing software projects is one of the major causes of inefficiency during development and testing. Larger programs, added features, and changing requirements usually leads to higher complexity and often with a large number of dependencies. The increased complexity can make it difficult for application developers to understand how various files and code segments work in relation to each other. It also becomes more difficult to predict the implications of modifying the codebase. While many interactive development environments (IDE) and standard development kits (SDKs) have built in error detection feature, some of these errors remain in the final product.

We intended to solve this problem through the use of software visualization, the presentation of information about a software system through static, interactive, or animated 2D/3D representations of their structure, execution, behavior, and/or evolution[10]. Since software projects encompass a very large scope, so we decided to focus on only Android applications. The backbone for Android programs is Java, and both this programming language and the Android framework have been developed, used, and tested thoroughly over the past decade. As Android users currently make up over 75% of the worlds smartphone users[3][7], it is important that developers are able to properly see and fix any vulnerabilities that may be found during the development and testing phases of their projects. The objective for our design project, the Androlyzer, was to construct a program that would analyze compiled software code, and graphically highlight areas of possible bugs and security vulnerabilities. Our program would have as its goal to increase usability and readability of security analysis tools and therefore improve productivity.

The Androlyzer begins by parsing the Android Application Package (APK file). An Android APK is the standard format used to distribute and install mobile android applications and middleware. Essentially a type of archive file based on a zip format called Jar, it is used as a container to aggregate all the Java class files of an application for distribution. The Androlyzer contains three third party applications and a javascript visualizer. The first application is called Dex2Jar, used to convert the apk in jar format. Once converted into a file with the extension .jar, we will analyze the files for both bugs and dependency issues and then feed this output into a Javascript visualizer to be displayed in a web browser for user to view. Bug analysis was done by FindBugs, while dependencies were checked with DependencyCheck.

Related programs to detect vulnerabilities in Android applications already exist, such as the University of Washingtons SPARTA project and Lins AndroBugs project. However, these programs were either difficult and cumbersome to setup or lacked visualization features as with AndroBugs. The key idea for our design was to draw on pre-existing methodologies for representing code and issues more visually and applying them to identify security vulnerabilities within a APK. We also drew on the concept of visual learning, used stylized lists, and addressed common readability of issues found in program code so users have an easier time understanding a bug reports.

The methodology for evaluating our design was based on evaluating the usability of an analysis program and the readability of the output bug reports. The results of the evaluation are based on a survey of two groups of people. One group used the Androlyzer to view any bugs and dependencies of a particular selected APK. The other group using FindBugs and DependencyCheck with no additional visualization. The survey results were pooled and compared between the two groups to determine if results of the Androlyzer improved comprehension of vulnerabilities within Android applications.

II. Related Works

Existing works for identifying program security flaws and malware exist, but there were not many that fit the goals that we had when creating our project. The SPARTA project, or Static Program Analysis for Reliable Trust Apps, is a program that was created to build a toolset that aimed at verifying security of mobile phone applications. SPARTA was proposed as a verification model for use in such app stores to guarantee that the apps are free of malicious information flows[8]. They provide tools in type-checking, allowing the user to specify a security property, annotate the source code with type qualifiers. However, this program did not have the visualization properties and the specific annotations and large documentation made it difficult for new users to learn and use.

AndroBugs is a Android vulnerability analysis system aimed at helping developers find potential security vulnerabilities in Android systems. Created by Yu-Cheung Lin, it had an interface that was command line based. Features for this framework included finding security vulnerabilities, checking if code was up to best practices, checking for dangerous commands, and checking an applications security protection[1]. Instructions were included for both Windows and Linux, but like SPARTA this work did not contain the visualization for the results. The report was displayed in the console of the terminal, which means the user often had to read through large amounts of text to identify security issues.

Many others have created toolsets with goals of improving security in mobile Android applications. A team led by Will Klieber created tools to address the issues of leakage of sensitive information from a sensitive source and another to address activity hijacking. Using the concept of information flow, they sought to address integrity concerns such that highly sensitive information does not flow to a place not authorized to receive the data. Activity hijacking occurs when a malicious app receives a message intended for another app. This can occur when a malicious app uses a confusing name that tricks the device into sending the message to someplace else. Kleibers et al created a second tool to help find the likely violation of secure coding rules in order to reduce the amount of activity hijacking[9]. There has been a lot of research, both past and ongoing, on static analysis for Android applications[6], but very few have the visualization properties we hoped to address with our project.

III. Adversary Model

Software security levels are relative to the powers of an adversary trying to compromise a system. When building security systems, the security level and competitiveness are usually compared against an adversary model. The specific adversary model for testing will depend on the what the system is designed to resist against. According to Wikipedia, there three common adversaries. The oblivious adversary is the weak adversary; they know the algorithm and code but do not know how to get the randomized results of the algorithm. The adaptive online adversary is the medium adversary who must make its own decision before it is allowed to know the decision of the algorithm. Finally, the adaptive offline adversary is the strong adversary who knows everything. Randomization does not help against it because they also know the random number generator.

For our system, the objectives of an adversary may include:

- Acquiring traffic information when users use an Android application
- Changing the app from trial (reduced-feature) mode to a full-featured version, causing the app developer to lose revenue [4]
- Relabelling the app to pass it off as their own [5]
- Extracting personal (user data) or proprietary information (API keys, security certificates) [5]

The initial capabilities of our adversary may include access to a copy of the APK and access to reverse engineering tools to decompile and view the files of the APK. With the ability to view source code, the adversary would be able to alter code segments and functionality that could negatively affect the usage of the app either against consumers or the developers themselves. During an attack, the capabilities include the ability to eavesdrop on wireless communication of someone else running that app, and possibly the temporary physical access to a users Android device if the app did not store data securely.

Based on these objectives and capabilities, our system is built to resist against a medium adaptive adversary. This adversary is able to view source code of an apk and exploit possible vulnerabilities to allow them to access or abstraction of private data, use the app for their own means, or perform other actions that may affect others. With our system, we have been able to detect these vulnerabilities ahead of time and provide fixes to ensure they did not make it to the final product, thus limiting the strength of the adversary.

IV. System Design

Android applications are packaged into a file the Android Application Package (APK) extension. As discussed earlier, it contains Java classes from the source code (compiled in a Dalvik Executable, or dex format) and resource files for the application. In order to analyze the files within the APK, we needed to convert the files to the proper format. Methods include changing the extension to a zip file and extracting the contents, or converting the APK to a jar format. Once converted to jar format, the jar file was used as an input for analysis. The output was an xml file that was read by a jquery script to parse and display the results in a readable format within a web browser.

Our design began by selecting a suitable program for converting APK files into jar. Dex2Jar was a simple command line interface program that suited our needs, taking either APK or dex files and converting them into jar files. For detecting security bugs and vulnerabilities in source code, we decided to go with FindBugs as they had an extensive library of documented issues and testing done since the time the project started. FindBugs is still being maintained, with the last update in October 2016. For checking dependencies within the apk, we went with OWASP Dependency Checker for its multiple support libraries. All three programs were also selected due to them each having a command line interface, as we decided to use bash scripts to integrate the three third party applications together. Therefore, the user would only need to one run script that would produce results for both security bugs and program dependencies. A general workflow of our system can be seen in Figure 1 of the Appendix.

The design decisions made were meant to help developers keep in mind several principles of designing secure systems. These principles included the questioning of assumptions, as analysis of imported libraries implies the user is does not assume that these libraries are secure; defense in depth as the application was analyzed from imported libraries up to written source code that used the libraries, which helps to enforce the need to have multiple layers of security; and open design as the visualization shows users that vulnerabilities remain exploitable even if the users attempt to hide them.

V. System Prototype

Our system consists of three third party applications tied together with a bash script: Dex2Jar-2.0, FindBugs-v3.0.1, and OWASP DependencyCheck. The final step and component was the visualization with the jquery script that read the xml output and displayed in a web browser for viewing. Firefox will be used as the default viewing browser.

Dex2Jar was a program created to work with android apk, dex, and java class files. While documentation for this project is limited, but the main goal is the conversion of the files in apk format into android .class files (zipped as jar). Use of Dex2Jar was simple; once extracted to a location the user simply needed to run ”sh dex2jar.sh [nameofapk.apk]” using the command line interface.

The program created as output a jar file named nameofapk-dex2jar.jar in the root directory of Dex2Jar.

FindBugs was a program created for security audits of Java web applications. It is an extensively tested plugin created for multiple platforms, including popular IDEs such as Eclipse and IntelliJ/Android Studio. It can detect 93 different vulnerability types with over 200 unique signatures, and includes multiple supports for Android frameworks and libraries[2]. It is an open source project that is still currently being updated. FindBugs was used to check for any type of security bugs and vulnerabilities that occurred.

The purpose of OWASPs DependencyCheck is to take advantage of the existing bug database compiled by the OWASP Foundation to check if imported libraries have known security flaws. The most important portion of this tool is the database of known bugs which is used by several known cyber security government agencies around the world, including Canadian Cyber Incident Response Centre and Center for Internet Security (CIS) in USA. The tool takes a jar file and scans for any dependencies.

Our project integrated all three third party applications by using multiple bash scripts written for Linux/Mac OSX. Separate bash scripts were first created for the Dependency-Check and FindBugs programs. One main bash script was then created that called each of those independently so that only one command was needed to start the entire program. At each step, the necessary xml output files were moved into a common folder. A javascript jquery parsing script was called by a python script at the end in order to format and display the results in the FireFox browser. A capture of the results displayed in the browser for Bugs and Dependencies are shown in Figure 2 and Figure 3 of the Appendix, respectively.

The idea of stylized lists was used as we felt that users have an easier time reading information organized into blocks. The color coding was also created in order for users to quickly identify which issues were of higher priority. Information about functions and where the specific vulnerability can be found within the Java files are also provided so the user was able to quickly search up and navigate to that particular area in the source code. Since the xml files are saved into a specified output folder, the user was also able to close the browser and reopen it for later use.

VI. System Evaluation

Evaluation Methodology

The evaluation of our system was based on a survey, refer to Figure 5 of the Appendix, of ten different people split into two groups. One group would attempt to view bugs and dependencies while using the Androlyzer, while the other were tasked with using the native programs separated. Each individual was chosen based on their experience and knowledge with computer science, but otherwise had little to no experience with the types of tools described earlier. The individuals were placed into the two groups (A and B) randomly.

For members of group A, they were instructed to attempt to install OWASP DependencyCheck and Findbugs and analyze a simple android application for security flaws and dependencies. The subject was required to answer a set of questions while doing the analysis pertaining to their experience in using the programs and locating these vulnerabilities within the code. Group B used the Androlyzer, and like Group A were asked to report on their experience in looking for the vulnerabilities within the source code running the system.

Results of the Evaluation

The evaluation of the Androlyzer compared to the original tools showed an improvement in the ease of installation and usability, refer to Figure 4 of the Appendix. The averaged responses showed that subjects found that the Androlyzer performed better than the original tools in terms of usability and readability.

Subjects from group A reported that using Dependency-Check and FindBugs was difficult due to lack of understanding of the console inputs in the tools documentation. Group A also reported that the resulting output xml output from FindBugs was difficult to follow citing difficulty tracking tags and end tags. All members of group A reported that they would not use this on a regular basis and rated the overall experience below five.

Subjects from group B found that the installation and use of the Androlyzer was generally easy and reported no issues. Three subjects reported that the colors used to differentiate severity of bugs could have been more distinct than multiple shades of red. Another issue was that some users did not realize that there were toggle buttons at the top to show either dependency vulnerabilities or source code vulnerabilities and were later informed that there were additional bugs hidden by the toggle. In general subjects found the collapsible bug descriptions useful for focussing on specific bugs.

Subjects from both groups reported that running the scripts took a very long time. This is attributed to DependencyCheck downloading the most updated catalog of known bugs. The tool will run faster for subsequent analysis as Dependency-Check will not have to redownload the catalog. As a result time was no longer taken into consideration as this was mostly dependent on the subjects internet download speed.

Discussion of the Results

The results shows that in terms of usability, the Androlyzer is an improvement. Several issues with the design of the Androlyzer were mentioned by group B, however these issues can be addressed easily. However the results of this survey may be flawed as the sample size of the survey is far too small to be representative of the Android user and developer base. Another issue with this survey was that it was conducted on people who the members of this group have prior relations with. Hence there is an inherent bias associated when asking a colleague to be surveyed while using a program. The influence of both of these issues could not be addressed adequately due to the time constraints for this project.

VII. Discussion

The Androlyzer makes existing tools more usable which is the motivation for the project. The GUI for the Androlyzer was finalized to a set of blocks and text elements as we felt that our original design of mapping out the code structure of an APK would prevent the user from understanding where vulnerabilities lie. We realized that rather than displaying a map of the entire code base, it would be better to show the bare minimum required to identify and locate a security bug and allow for the option to expand a block for further information.

With regards to our adversary model, we believe that the Androlyzer is a good starting point for the analysis of APKs as each patched bug reduces the likelihood of a security breach, however the Androlyzer is inherently limited and provides no further analysis than the tools it uses.

Our design benefits by making it easy to include more tools which use console commands. As a result, existing work such as AndroBugs, can be included for a more generalized bug checking tool. The Androlyzer could also be expanded to include other languages such as C or Python as long as the tools have command line interfaces. However the Androlyzer is limited to mostly text-based tools and cannot interface with tools that only have a GUI version available. Another limitation that the Androlyzer currently faces is the inability to specify specific flags for the individual tools. This limitation can be fixed by editing the scripts directly to suit the users needs however this reduces the usability of the Androlyzer. A solution would be to implement an advanced settings GUI for the Androlyzer to allow users to enable and disable settings for the integrated tools.

The results of the evaluation were as expected where the Androlyzer performed better than the original tools in all categories. This was not surprising as the Androlyzer automates the majority of the steps required to do this analysis and that it hides the majority of the output until the user deliberately selects to view it, removing visual clutter. The results are still considered biased and not fully representative of the Android user and developer base however the consistency of the results are promising.

VIII. Conclusion

Android code can become difficult to read for a developer and this can result in security bugs. The purpose of the Androlyzer is to remedy this issue and provide concise and accurate information to a developer. This was done by integrating FindBugs and DependencyCheck into a single program to search for security bugs in the source code as well as any dependencies used. The results of the analysis would then be displayed in collapsible blocks. The results of our evaluation show an improvement in usability over the original tools and the design of our project allow for further integration with other tools or even integration with IDEs.

Android currently makes up a large share of the smartphone market and hence there is a large number of people who are potentially at risk of a security bug. While many tools do exist to scan for security bugs, many of them are not easy to use and can confuse developers. Our tool is aimed at newer developers and power users to allow them to more easily understand what and where bugs exist in the applications they are developing or using.

IX. Appendix

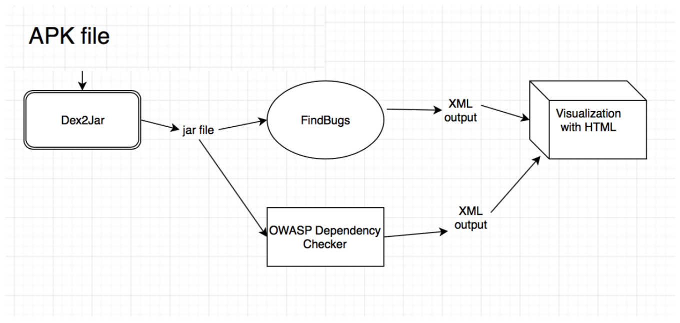
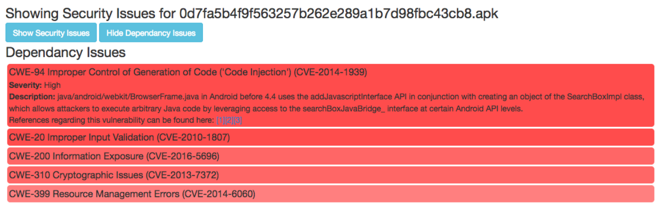


Figure 1: System work flow from APK file to XML output for visualization.



Showing Security Issues for 0d7fa5b4f9f563257b262e289a1b7d98fbc43cb8.apk

Dependancy Issues

CWE-94 Improper Control of Generation of Code ('Code Injection') (CVE-2014-1939)
Severity: High
Description: java/android/webkit/BrowserFrame.java in Android before 4.4 uses the addJavascriptInterface API in conjunction with creating an object of the SearchBoxImpl class, which allows attackers to execute arbitrary Java code by leveraging access to the searchBoxJavaBridge_ interface at certain Android API levels.
References regarding this vulnerability can be found here: [1][2][3]

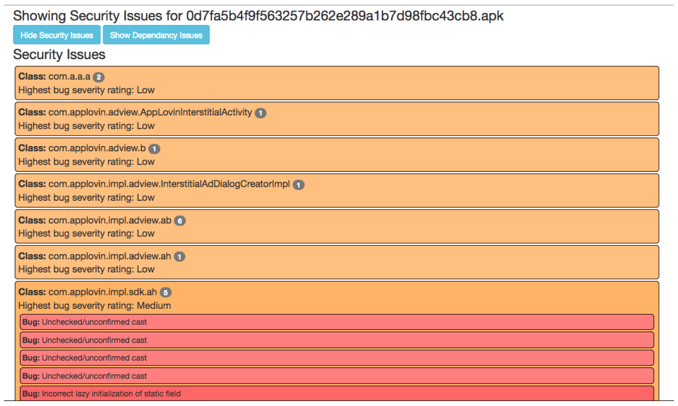
CWE-20 Improper Input Validation (CVE-2010-1807)

CWE-200 Information Exposure (CVE-2016-5696)

CWE-310 Cryptographic Issues (CVE-2013-7372)

CWE-399 Resource Management Errors (CVE-2014-6060)

Figure 2: Sample Bug output from our integrated program, the Androlyzer.



Showing Security Issues for 0d7fa5b4f9f563257b262e289a1b7d98fbc43cb8.apk

Security Issues

Class: com.a.a.a 2
Highest bug severity rating: Low

Class: com.applovin.adview.AppLovinInterstitialActivity 1
Highest bug severity rating: Low

Class: com.applovin.adview.b 1
Highest bug severity rating: Low

Class: com.applovin.impl.adview.InterstitialAdDialogCreatorImpl 1
Highest bug severity rating: Low

Class: com.applovin.impl.adview.ab 6
Highest bug severity rating: Low

Class: com.applovin.impl.adview.ah 1
Highest bug severity rating: Low

Class: com.applovin.impl.sdk.ah 5
Highest bug severity rating: Medium

Bug: Unchecked/unconfirmed cast
Bug: Unchecked/unconfirmed cast
Bug: Unchecked/unconfirmed cast
Bug: Unchecked/unconfirmed cast
Bug: Incorrect lazy initialization of static field

Figure 3: Sample Dependency output from our integrated program, the Androlyzer.

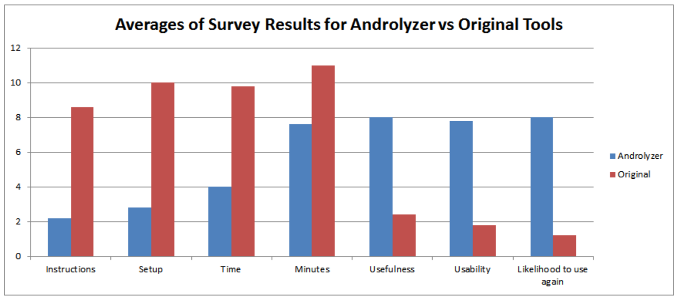
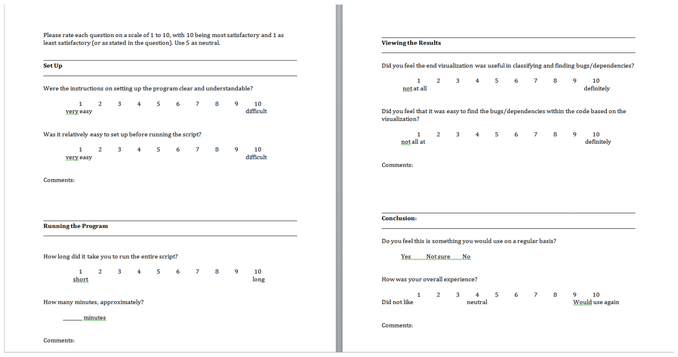


Figure 4: Survey results averaged and compared between the Androlyzer and DependencyCheck with FindBug.



Please rate each question on a scale of 1 to 10, with 10 being most satisfactory and 1 as least satisfactory (or as stated in the question). Use 5 as neutral.

Set Up

Were the instructions on setting up the program clear and understandable?

```
   1    2    3    4    5    6    7    8    9    10
very easy                                    difficult
```

Was it relatively easy to set up before running the script?

```
   1    2    3    4    5    6    7    8    9    10
very easy                                    difficult
```

Comments:

Running the Program

How long did it take you to run the entire script?

```
   1    2    3    4    5    6    7    8    9    10
 short                                         long
```

How many minutes, approximately?

_______ minutes

Comments:

Viewing the Results

Did you feel the end visualization was useful in classifying and finding bugs/dependencies?

```
   1    2    3    4    5    6    7    8    9    10
not at all                                   definitely
```

Did you feel that it was easy to find the bugs/dependencies within the code based on the visualization?

```
   1    2    3    4    5    6    7    8    9    10
not all at                                   definitely
```

Comments:

Conclusion:

Do you feel this is something you would use on a regular basis?

Yes   Not sure   No

How was your overall experience?

```
           1    2    3    4    5    6    7    8    9    10
Did not like             neutral                 Would use again
```

Comments:

Figure 5: Survey questions for evaluation of the Androlyzer versus Dependency-Check and FindBugs.

REFERENCES

[1] A. B., "AndroBugs/AndroBugs_Framework," GitHub. [Online]. Available: https://github.com/AndroBugs/AndroBugs_Framework/blob/master/README.md. [Accessed: 03-Dec-2016].

[2] Find Security Bugs, Home - Find Security Bugs. [Online]. Available: https://find-sec-bugs.github.io/. [Accessed: 03-Dec-2016].

[3] Hachman, Mark (2016-02-18). Android leads, Windows phones fade farther in Gartners smartphone sales report. PCWorld from IGD. Retrieved 2016-11-04. [Accessed: 2016-11-08].

[4] H. Hira, "5 Apps to Hack In-App Purchase in Android", Ultimatepctech.com, 2016. [Online]. Available: http://www.ultimatepctech.com/2016/09/5-apps-to-hack-in-app-purchase-in.html. [Accessed: 2016-11-08].

[5] "How to Hack a Mobile App: It's Easier than You Think!", Security Intelligence, 2016. [Online]. Available: https://securityintelligence.com/how-to-hack-a-mobile-app-its-easier-than-you-think/. [Accessed: 2016-11-08].

[6] M. D. Ernst, S. Han, P. Vines, E. X. Wu, R. Just, S. Millstein, W. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. B. Barros, and R. Bhoraskar, Collaborative Verification of Information Flow for a High-Assurance App Store, Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security - CCS '14, 2014.

[7] P. Faruki et al., "Android Security: A Survey of Issues, Malware Penetration, and Defenses," in IEEE Communications Surveys & Tutorials, vol. 17, no. 2, pp. 998-1022, Secondquarter 2015. http://ieeexplore.ieee.org.ezproxy.library.ubc.ca/stamp/stamp.jsp?tp=&arnumber=6999911&isnumber=7110413

[8] SPARTA! Static Program Analysis for Reliable Trusted Apps, SPARTA! Static Program Analysis for Reliable Trusted Apps. [Online]. Available: http://types.cs.washington.edu/sparta/current/sparta-manual.html. [Accessed: 03-Dec-2016].

[9] Two Secure Coding Tools for Analyzing Android Apps, SEI Insights, 2014. [Online]. Available: https://insights.sei.cmu.edu/sei_blog/2014/04/two-secure-coding-tools-for-analyzing-android-apps.html. [Accessed: 03-Dec-2016].

[10] Wikipedia (2016). Software Visualization. Wikipedia. Retrieved 2016-11-04.