DroidPatrol: A Static Analysis Plugin For Secure Mobile Software Development

Md Arabin Islam Talukder[1], Hossain Shahriar[1], Kai Qian[1], Mohammad Rahman[2], Sheikh Ahamed[3], Fan Wu[4], Emmanuel Agu[5]
[1]Kennesaw State University, Marietta, GA, USA
[2]Florida International University, Miami, FL, USA
[3]Marquette University, Milwaukee, WI, USA
[4]Tuskegee University, Tuskegee, AL, USA
[5]Worcester Polytechnic Institute, Worcester, MA, USA
mtalukd1@students.kennesaw.edu, {hshahria, kqian}@kennesaw.edu, marahman@fiu.edu
sheikh.ahamed@mu.edu, fwu@tuskegee.edu, emmanuel@wpi.edu

***Abstract* - While the number of mobile applications are rapidly growing, these applications are often coming with numerous security flaws due to the lack of appropriate coding practices. Security issues must be addressed earlier in the development lifecycle rather than fixing them after the attacks because the damage might already be extensive. Early elimination of possible security vulnerabilities will help us increase the security of our software and mitigate or reduce the potential damages through data losses or service disruptions caused by malicious attacks. However, many software developers lack necessary security knowledge and skills required at the development stage, and Secure Mobile Software Development (SMSD) is not yet well represented in academia and industry. In this paper, we present a static analysis-based security analysis approach through design and implementation of a plugin for Android Development Studio, namely DroidPatrol. The proposed plugins can support developers by providing list of potential vulnerabilities early.**

Keywords-Android, Secure software development, Static analysis, Tainted data flow, SQL Injection.

I. INTRODUCTION

With the increased demands of mobile applications in recent years, we have also witnessed numerous major cyber-attacks, resulting in stolen personal credit card numbers, leakage of classified information vital for national defense, industrial espionage resulting in major financial losses, and many more consequences. Hackers have managed to make secure computing a more difficult task. Therefore, there is a greater need for not only including the concept of cybersecurity but also the secure software development in the training of computer science, information technology, and related field professionals. The rapid growth of mobile computing also results in a shortage of professionals for mobile software development, especially for Secure Mobile Software Development (SMSD) professionals, and insufficient tool support to develop secure mobile applications [12, 13, 14].

If all or most of the possible vulnerabilities can be addressed and fixed for a mobile software during its development phase, the potential attack space will be minimized. Many open source static Java code analysis tools help developers to maintain and clean up the code through the analysis performed without actually executing the code such as Eclipse IDE [15], IntelliJ IDE [16], and FindBugs Plugin [17]. These tools focus on finding probable bugs such as inconsistencies, helping improve the code structure, conform source code to guidelines, and provide quick fixes. The security vulnerability checking is not their major task.

Source code analysis tools, also referred to as Static Application Security Testing (SAST) Tools, are designed to analyze source code and to help to find security flaws with a high confidence that what's found is indeed a flaw (readers can see the survey [8] for list of exhaustive state-of-the art tools). However, there is no tool that can just automatically finds all flaws and can guarantee all detecting are positive or never miss any potential flaws [18]. Currently, there is no tool available that would allow mobile application developers to analyze their project source code for detecting security flaws within the development environment (e.g., Android Development Studio).

In this paper, we design and implement DroidPatrol, a plugin to be integrated with the Android Development Studio to perform tainted data flow-based static analysis. DroidPatrol allows developers to specify a list of sources and sinks and enable them to see the possible paths within the source code and suggestion of corresponding fixes.

This paper is organized as follows. Section II discusses related work. Section III provides overview of tainted data flow analysis, followed by the design of DroidPatrol. Finally, Section IV concludes the paper.

II. BACKGROUND AND RELATED WORK

A. Background on Static anlaysis

Static program analysis generally involves an automated tool that takes as input the source code (or object code in some cases) of a program, examines this code without executing it, and yields results by checking the code structure, the sequences of statements, and how variable values are processed throughout the different

978-1-7281-2607-4/19/$31.00 ©2019 IEEE
DOI 10.1109/COMPSAC.2019.00087

function calls. The main advantage of static analysis is that all the code is analyzed [8]. This differs from dynamic analysis where portions of code could only be executed under some specific conditions that could never be met during the analysis phase. A typical static analysis process starts by representing the analyzed app code to some abstract models (e.g., call graph, control-flow graph, or UML class/sequence diagram) based on the purpose of analysis.

A control-flow analysis is a technique to show how hierarchical flow of control within a given program are sequenced, making all possible execution paths of a program analyzable [8]. A data-flow analysis [31] is a technique to compute at every point in a program a set of possible values. This set of values depends on the kind of problem that has to be solved using data-flow analysis. The analysis allows us to identity the set of definitions reachable at every program point. A program usually starts with a single entry point. A quick inspection of the main entry method's code can list the method(s) that it calls. Then, iterating this process on the code of the called methods leads to the construction of a directed graph, commonly known as the call graph in program analysis. Our work relies on the call graph generated by Soot analyzer [5], which is a popular tool also used by others.

B. Related work

Below, we discuss a number of commonly available tools for static and dynamic analysis of android applications. None of them allows developers to integrate the tool in Android Development Studio to streamline the identification of common Android Security bugs and fix them early. Readers are suggested to see the detailed survey [8] for other available tools for statically analyzing Android Applications for security bug identification.

FlowDroid is an open source Java based static analysis tool that can be used to analyze Android applications for potential data leakage. FlowDroid is a context, object sensitive, field, flow, and static taint analysis tool that specifically models the full Android lifecycle with high precision and recall [23]. The tool can detect and analyze data flows, specifically an Android application's bytecode, and configuration files, to find any possible privacy vulnerabilities, also known as data leakage [24]. It is not intended to analyze malware [19]. However, it cannot find common security bugs in Android such as SQL Injection, output encoding, Intent leakage, and lack of secure communication.

Cuckoo is a widely used malware analysis tool based on dynamic analysis (i.e., it runs an application under test in a controlled emulator). It is capable of methodically examining multiple variants of Android malware applications through controlled execution into virtual machines that monitor the behaviors of the applications [21]. It comprises a host that is responsible for the sample execution and the analysis in which the guests run. When the host has to launch a new analysis, it chooses the guests and uploads that sample as well as the other components that are required by the guest to function [2O]. Once the analysis has completed, the analyzer refers the results of the analysis to the ResultServer, which in turn will implement whichever processing modules are configured (the modules used to populate the product of the analysis, the report) and produce the report [22].

Yanick et al. [11] detected logic bombs in Android applications using a number of static analysis tools, including FlowDroid, Kirin, TriggerScope, and DroidAPIMiner. A logic bomb is an unauthorized software that changes the output of the Android application or does applications actions that are not intended. Among the other analysis tools, FlowDroid had the highest false positive percentage, and second lowest false negative percentage.

The DroidSafe project [9] developed effective program analysis techniques and tools to uncover malicious code in Android mobile applications. The core of the system is a static information flow analysis that reports the context under which sensitive information is used. For example, Application A has the potential to send location information to network address. DroidSafe reports potential leaks of sensitive information in Android applications. It still suffers from imprecision due to 1) unacceptable numbers of false positive alarms, and 2) the use of unsound techniques that may leave errors uncovered.

Many other efforts have been made to enhance the secure software development in recent years. Application Security IDE (ASIDE) plug-in for Eclipse can warn programmers of potential vulnerabilities in their code and assists them in addressing these vulnerabilities. The tool is designed to improve awareness and understanding of security vulnerabilities and to increase utilization of secure programming practices. ASIDE addresses input validation vulnerabilities, output encoding, authentication and authorization, and several race condition vulnerabilities [1-3]. However, it cannot identify Android specific security flaws. Further, ASIDE only works in the Java Eclipse IDE and cannot support Android Development Studio.

Android has a complex communication system for sharing and sending data in both inter and intra applications. Simple static analysis usually cannot satisfy further requirement. Malicious applications may take advantage of built-in feature (e.g., Intent object broadcast by victim applications can be intercepted by a malware running on the same device) to avoid detection. Recently many tools are developed to perform taint-based static analysis checking, like Findbugs and DidFail [1O]. They are not capable of detecting all known Android security bugs based on OWASP guidelines [7]. Detection of potential taint flows can be used to protect sensitive data, identify leaky apps, and identify malware.

III. DroidPatrol Design and Implementation

DroidPatrol (see Figure 1) is designed based on Soot [5] and Jimple [4]. Soot is one of the most used static analyzers for Java-based applications. Android application is based on Java, so we used some basic concept and static analysis library APIS of Soot. Apart from Soot, DroidPatrol requires an input application file (e.g., app-debug.apk),

dependent libararies (e.g., android.jar) and a list of sources and sinks (e.g., SourceAndSink.txt) file to perform the flow analysis. It first decompiles the input apk file followed by generating and analyzing call graphs (between method definition and method call locations) to find out possible data leakages.

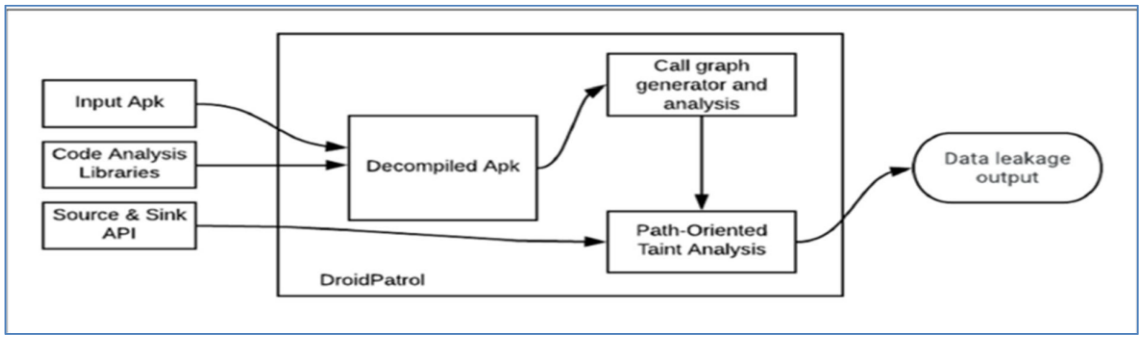

Figure 1: Architecture of DroidPatrol

A. Scope of DroidPatrol

DroidPatrol plugin is intended to identify Android security bugs based on the on most current OWASP 2O17 mobile top 1O mobile security risks [7] for the category of SQL injection, unintended data leakage, insecure communication, insecure data storage vulnerability detectors. Table 1 shows examples of data flow leakage, list of sources and sinks for extraneous functionality, improper platform usage, insecure data storage, insecure communication, and insecure authorization. DropidPatrol can recognize SQL injection vulnerability and data leakage in mobile applications, which may face the threat of potential malicious code injection, and then issue a warning on the code line. Following the provided options, developers can enforce a new secure statement to replace the unsecure statement. A built package can also be loaded into the Android Studio IDE, which will result in parsing Android java source code, identifying specific API calls, warning potential vulnerabilities, recommending code statements for replacement.

Table 1: Example of Detector, Sources and Sinks

Example of Flow Detector	Source (example)	Sink (example)
Extraneous Functionality	Bundle class and Intent class	Log class
Improper Platform Usage	View class	HTTP class
Insecure Data Storage	SQLite database class, Shared Preferences class	Rest API, SmsManager class
Insecure Communication	Intent, Bundle class	Broadcast class
Insecure Authorization	EditText class	Backend Rest Api, SmsManager class

For many Android security vulnerabilities and flaws on the top 1O mobile risks by OWASP and other new identified unlisted flaws we need to develop our own customized detectors. A practical challenge in static analysis is to control the rate of false alarms while not missing any (potentially dangerous) behaviors of applications. This is especially significant due to a number of features of Android.

First, Android is an event-based system. The control flow is driven by events from an application environment that can trigger various method calls. How to capture all the possible control flow paths in this open and reactive system while not introducing too many spurious paths (false alarms) is a significant challenge.

Second, the Android runtime consists of a large base of library code that an app depends upon. The event-driven nature makes a large portion of the control-flow involve the Android library. While fully analyzing the whole library code could improve the analysis' faithfulness, it may also be prohibitively expensive (or imprecise).

Third, Android is a component-based system and makes extensive use of inter-component communication (ICC). For example, a component can send an Intent to another component. The target of an Intent could be specified explicitly in the Intent or be implicit and decided at runtime. Both control and data can flow through the ICC mechanism from one component to another. Capturing all ICC flows accurately is a major challenge in static analysis. Before we discuss our design (in section C) addressed these challenges, we provide a working example of SQL Injection detection using DroidPatrol in the next section.

B. An Example Application – Data Leak Detection with DroidPatrol

SQL injection is a common security vulnerability in mobile applications leading to data leakage. It works by adding user supplied data to a query string which leads to the alteration of SQL queries leading hackers to access to unauthorized data and bypassing logins. SQL Injection is usually used to attack Web Views or a web service. However, it can also be used to attack Activities in Android applications.

Consider the code segment in Figure 2. Here, a SELECT query is formed with user provided user id (*uid*) and password (*pwd*) variables in the method *isValidUser()*.

The input is obtained from *username* and *password* text boxes (which we mark as source) in the *onCreate()* method. After the query runs, the output is the name and grade, which are displayed by setting as text to textboxes (data sink) in the *isValidUser()* method.

```
public void onCreate(Bundle savedInstanceState){
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main);
        username = findViewById(R.id.textView1); // source
        password = findViewById(R.id.textView1); // source
}
public boolean isValidUser(){

   uid = findViewById(R.id.editText1); //retrieving id
   pwd = findViewById(R.id.editText2); //retrieving password

   String qry = "select name, grade from users where user_id= '"
+ uid + "' and password = '" + pwd +"'";
   SQLiteDatabase db;
    ...
   Cursor c = db.rawQuery (qry, null );
   name.setText(c.getString(columindex1)); //sink
   grade.setText(c.getString(columnindex2)); //sink
   return c.getCount() != O;
}
```

Figure 2: Example of vulnerable code

Assume, user id is "jdoe", pwd is "secret". Then, the *qry* is select * from users where password= 'jdoe' and pwd ='secret'. Since, the input is not filtered, an attacker can exploit the application by providing malicious inputs for uid value as *' or 1=1 –* , whereas pwd value as blank. The qry would be select * from users where password='' or 1=1 -- 'and pwd=''. The -- symbol means comment by query engine. Thus, the query is changed to *select * from users where password='' or 1=1*. This query will be evaluated as true and select all information (name and grade) from the table users as opposed to one entry matching with the *uid* and *pwd*. This way, an attacker can bypass authentication.

To determine the tainted data flow from every possible point of a program, dataflow analysis can be used. Data flow analysis works on a fixed abstraction and the outcomes are often a) flat to symbolic over-approximation, and b) do not show instance of traces defining paths from the origin to sinks for a given vulnerability [6].

We first define the sources and the sinks (see Figure 3). Source means location where input data may be obtained from external inputs such as a user or database query. For example, in Figure 3, the source is defined as database Cursor object. This object allows a program to retrieve data. Data obtained from source can be transferred to a third party via SMS messaging. In Android, to send an SMS message, SmsManager object can be used which subsequently requires SEND_SMS permission to be listed in the manifest file. Figure 2 shows both SmsManager class and SEND_SMS permission listed in the sink list. A developer can include other possible sources and sinks based on secure programming practices and OWASP guidelines. This allows the flexibility to not only detecting new security bugs, but also reducing false positive warning.

```
<android.database.Cursor: java.lang.String getString(int)> ->
_SOURCE_

<android.telephony.SmsManager: void
sendTextMessage(java.lang.String,java.lang.String,java.lang.String,a
ndroid.app.PendingIntent,android.app.PendingIntent)>
android.permission.SEND_SMS -> _SINK_
```

Figure 3: Source and sink definition

The tool provides us a list of dataflow where information flow between sources and sinks are displayed in the log output of the Android IDE (Figure 4).

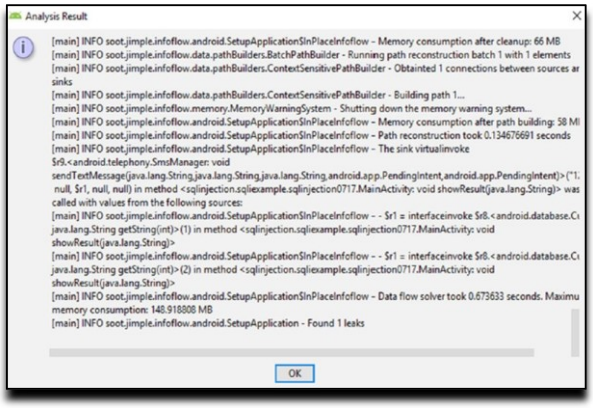

Figure 4: A sample result from DroidPatrol analysis

We show an example of highlighted code segment in the project source code when enabling the DroidPatrol. Moving the cursor on the highlighted method calls provide suggestions to fix the code for input validation before reaching to the sinks. Figure 5 shows that DroidPatrol detects code fragment related to data leak. It also shows the summarized output of the data flow analysis. It provides a list of possible sinks of the data then looks for possible source for each sink.

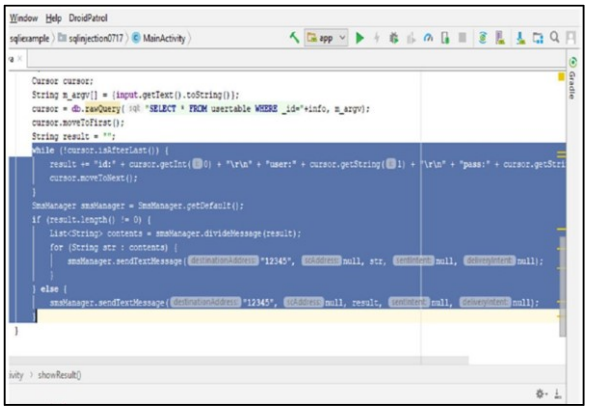

Figure 5: A Sample screenshot of DroidPatrol plugin highlighting vulnerable code

IV. Conclusions

Currently, there is no available plugins for Android Development Studio that can be integrated for static data

flow analysis. In this paper, we developed a plugin tool named DroidPatrol. We plan to make the tool open source for public use in the near future. The plugin can perform tainted data flow analysis of application as developers implement mobile applications and wish to detect various security bugs leading to privacy and data leaks based on OWASP guidelines. Our tool can highlight the code that should be paid attention for removing bugs. The plugin is lightweight as it integrates seamlessly with Android Studio without intensively consuming more system resources.

V. ACKNOWLEDGEMENTS

The work is partially supported by the National Science Foundation under award: NSF proposal 1723586, 1723578, 1636995, 1438858, and KSU OVPR Award 2018-19. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Michael Whitney, Heather Richter Lipford, Bill Chu, and Jun Zhu. Embedding Secure Coding Instruction into the IDE: A Field Study in an Advanced CS Course. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15)*, Minneapolis, MN, USA, 2015, pp. 60-65

[2] Michael Whitney, Heather Richter Lipford, Bill Chu, and Tyler Thomas. "Embedding Secure Coding Instruction into the IDE: Complementing Early and Intermediate CS Courses with ESIDE" In press, *Journal of Educational Computing Research*, 2017

[3] J. Xie, H. Lipford, B. Chu, Evaluating interactive support for secure programming, *Proceeding of SIGCHI Conference on Human Factors in Computing Systems*, Austin, TX, 2012, pp. 2707-2716.

[4] P. Pominville, Using Jimple Parse, March 200, https://www.sable.mcgill.ca/soot/tutorial/jimpleParser/index.html

[5] Soot Java Optimization Framework, Accessed from http://sable.github.io/soot/

[6] Katerina Goseva-Popstojanovaa, Andrei Perhinschib, On the capability of static code analysis to detect security vulnerabilities, community.wvu.edu/~kagoseva/Papers/IST-2015.pdf

[7] Projects/OWASP Mobile Security Project - Top Ten Mobile Risks, https://www.owasp.org/index.php/Projects/OWASP_Mobile_Security_Project_-_Top_Ten_Mobile_Risks

[8] Li Li, Tegawend'e F. Bissyand'e, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartela, Damien Octeauc, Jacques Kleina, Yves Le Traona, "Static Analysis of Android Apps: A Systematic Literature Review", Information and Software Technology, Volume 88, August 2017, Pages 67-95.

[9] DroidSafe, https://mit-pac.github.io/droidsafe-src/

[10] Karan Dwivedi Hongli Yin Pranav Bagree Xiaoxiao Tang Lori Flynn William Klieber William SnavelyDidFail: Coverage and Precision Enhancement

[11] Fratantonio, Y., Bianchi, A., Robertson, W., Kirda, E., Kruegel, C., & Vigna, G. (2016). TriggerScope: Towards Detecting Logic Bombs in Android Applications. 2016 IEEE Symposium on Security and Privacy (SP). doi:10.1109/sp.2016.30

[12] Hossain Shahriar, Kai Qian, Md Arabin Islam Talukder, Nidhibahen Patel and Dan Lo, Mobile Software Security Risk Assessment with Program Analysis, *Proc. of the 23rd IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, Taipei, Taiwan, December 2018, 2 pp.

[13] Kai Qian, Dan Lo, Hossain Shahriar, Lei Li, Fan Wu, Prabir Bhattacharya, Learning database security with hands-on mobile labs, Proc. of IEEE Frontiers in Education Conference (FIE), Oct 2017, pp. 1-6.

[14] Kai Qian, Hossain Shahriar, Fan Wu, Lixin Tao, Prabir Bhattacharya, Labware for Secure Mobile Software Development (SMSD) Education, Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education, March 2017, pp. 375-375.

[15] Eclipse IDE, https://www.eclipse.org/ide/

[16] IntelliJ IDEA, https://www.jetbrains.com/idea/

[17] FindBugs in Java Programs, http://findbugs.sourceforge.net/

[18] Hossain Shahriar et al., Mitigating program security vulnerabilities: Approaches and challenges, ACM Computing Surveys (CSUR), Vol. 44, Issue 3, Article 11, June 2012.

[19] Fratantonio, Y., Bianchi, A., Robertson, W., Kirda, E., Kruegel, C., & Vigna, G. (2016). TriggerScope: Towards Detecting Logic Bombs in Android Applications. 2016 IEEE Symposium on Security and Privacy (SP). doi:10.1109/sp.2016.30

[20] Underwood, K., & Locasto, M. E. (2016). In Search of Shotgun Parsers in Android Applications. 2016 IEEE Security and Privacy Workshops (SPW), 140-155. doi:10.1109/spw.2016.41

[21] What is Cuckoo? — CuckooDroid v1.0 Book. (n.d.). Retrieved from https://cuckoo-droid.readthedocs.io/en/latest/introduction/what/

[22] Installation — CuckooDroid v1.0 Book. (n.d.). Retrieved from https://cuckoo-droid.readthedocs.io/en/latest/installation/

[23] Golam Sarwar Babil ; Olivier Mehani ; Roksana Boreli ; Mohamed-Ali Kaafar. (2013). On the effectiveness of dynamic taint analysis for protecting against private information leaks on Android-based devices. *2013 International Conference on Security and Cryptography (SECRYPT)* (pp. 1-8). Reykjavik, Iceland: IEEE.

[24] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Mcdaniel, P. (2013). FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 14, 259-269. doi:10.1145/2594291.2594299