Mobile Software Security with Dynamic Analysis

Hossain Shahriar[1], Kai Qian[2], Md Arabin Islam Talukder[1], Dan Lo[2], Nidhibahen Patel[2]
[1]Department of Information Technology
[2]Department of Computer Science
Kennesaw State University, USA
{hshahria, kqian, dlo2}@kennesaw.edu
{mtalukd1, npate154}@students.kennesaw.edu

***Abstract*- The majority of malicious mobile attacks take advantage of vulnerabilities in mobile software (applications), such as sensitive data leakage, unsecured sensitive data storage, data transmission, and many others. Most of these vulnerabilities can be detected by analyzing the mobile software. In this paper, we describe a tainted dataflow approach to detect mobile software security vulnerability, particularly, SQL Injection.**

Keywords: Mobile software, Android security, SQL Injectionn, Tainted data flow analysis.

I. INTRODUCTION

With increasing demands of mobile applications in recent years, there is proportional growth in security threats to these mobile devices. A report from Trendmicro suggests that mobile ransomware increased by 415% in 2017 [2]. Hackers have managed to make secure computing a more difficult task.

The majority of malicious mobile attacks take advantage of vulnerabilities in mobile applications, such as sensitive data leakage via inadvertent or side channel, unsecured sensitive data storage, data transmission, and many others. Most of these vulnerabilities can be detected during development phase. However, most development teams often have virtually no time to address them due to critical project deadlines [3]. To combat this, the more defect removal filters there are in the software development life cycle, the fewer defects that can lead to vulnerabilities will remain in the software product when it is released. More importantly, early detection of defects enables the organization to take corrective action early in the software development life cycle [4].

The vulnerability assessment considers the potential impact and loss (confidentiality, integrity, availability) exploiting the weakness from a successful attack. Vulnerability assessment can be performed by analyzing code for the presence of data flows or API calls that may represent security risks and could be exploited with malicious inputs. Analysis detects flaws in the code early in the process, weaknesses can be fixed before hackers detect and exploit them [5]. In this paper, we apply tainted dataflow approach to detect SQL Injection vulnerability.

II. SQL INJECTION

SQL injection is a common vulnerability in mobile applications [5]. 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, but it can also be used to attack Activities in Android applications.

Consider the code segment in Figure 1. 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); //data source
        password = findViewById(R.id.textView1); //data source
}

public boolean isValidUser(){
   …
   uid = findViewById(R.id.editText1); //retrieving user provided id
   pwd = findViewById(R.id.editText2); //retrieving user provided
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)); //data sink
   grade.setText(c.getString(columnindex2)); //data sink
   return c.getCount() != 0;
}
```

Figure 1: 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 actual query gets changed to select * from

2473-3105/18/$31.00 ©2018 IEEE
DOI 10.1109/PRDC.2018.00039

users where password=" or 1=1. This query will be evaluated as true and select all information from users table as opposed to one entry matching with uid and pwd. This way, an attacker can bypass authentication.

III. Tainted Data Flow Analysis for SQL Injection

To determine the destination of tainted data 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 applied the Flowdroid tool [1] to detect SQL Injection (as shown in Figure 1). We first define the sources and the sinks (Figure 2). Source means location where input data may be obtained. For example, edit text field could be a data source. To use it as a source, we declare the base class name of "findViewById(int)" method which is "android.view.View" class. We need to declare the use of the source in the application. Here, in this example, the source and sink both have been used in the Activity class. Sink is the final destination of the data. In this example, we used source to find out the data from the database then finally set the data in a Textview, which is the sink of the data. Figure 2 shows the way of declaration of source and sink in a text file. Here, both source and sink have been used in the Activity class.

```
<android.app.Activity: android.view.View
findViewById(int)> -> _SINK_

<android.app.Activity: android.view.View
findViewById(int)> -> _SOURCE_
```

Figure 2: Source and sink definition

The tool provides us a list of dataflow where information leakage happens in command line (Figure 3).

```
[main] INFO soot.jimple.infoflow.android.SetupApplication$InPlaceInfoflow - The
sink $r2 = virtualinvoke $r0.<sqlinjection.sqliexample.sqlinjection0717.MainActi
vity: android.view.View findViewById(int)>(2131165304) in method <sqlinjection.s
qliexample.sqlinjection0717.MainActivity: void onCreate(android.os.Bundle)> was
called with values from the following sources:
[main] INFO soot.jimple.infoflow.android.SetupApplication$InPlaceInfoflow - - $r
2 = virtualinvoke $r0.<sqlinjection.sqliexample.sqlinjection0717.MainActivity: a
ndroid.view.View findViewById(int)>(2131165232) in method <sqlinjection.sqliexam
ple.sqlinjection0717.MainActivity: void onCreate(android.os.Bundle)>
[main] INFO soot.jimple.infoflow.android.SetupApplication$InPlaceInfoflow - Data
 flow solver took 0.349459 seconds. Maximum memory consumption: 94.541576 MB
[main] INFO soot.jimple.infoflow.android.SetupApplication - Found 1 leaks
Arabins-MacBook-Pro:FlowDroid-2.6.1 arabin$
```

Figure 3: A screenshot of the Fowdroid output

We also receive an xml output file that highlights the source and the sinks (Figure 4).

```
<?xml version="1.0" encoding="UTF-8"?>
<DataFlowResults FileFormatVersion="101">
<Results>
 <Result>
   <Sink Statement="$r2 = virtualinvoke
$r0.&lt;sqlinjection.sqliexample.sqlinjection0717.MainActivity:
android.view.View findViewById(int)&gt;(2131165304)"
  Method="&lt;sqlinjection.sqliexample.sqlinjection0717.MainActivity:
void onCreate(android.os.Bundle)&gt;">
   <AccessPath Value="$r0"
Type="sqlinjection.sqliexample.sqlinjection0717.MainActivity"
TaintSubFields="true">
    <Fields>
     <Field
Value="&lt;sqlinjection.sqliexample.sqlinjection0717.MainActivity:
android.widget.EditText input&gt;"
Type="android.widget.EditText"></Field>
    </Fields>
   </AccessPath>
 </Sink>
 <Sources>
   <Source Statement="$r2 = virtualinvoke
$r0.&lt;sqlinjection.sqliexample.sqlinjection0717.MainActivity:
android.view.View findViewById(int)&gt;(2131165232)"
Method="&lt;sqlinjection.sqliexample.sqlinjection0717.MainActivity:
void onCreate(android.os.Bundle)&gt;">
    <AccessPath Value="$r2" Type="android.view.View"
TaintSubFields="true">
    </AccessPath>
   </Source>
 </Sources>
</Result>
</Results>
</DataFlowResults>
```

Figure 4: XML output from Flowdroid between source and sink

Figure 4 shows the output of the data flow analysis by Flowdroid. It provides a list of possible sinks of the data then looks for possible source for each sink. In the xml file, we can see that the sink and source present in the source code (Figure 1) were found.

Acknowledgement

The work is partially supported by the National Science Foundation Award: proposal# 1723578.

References

[1] FlowDroid, https://github.com/secure-software-engineering/FlowDroid
[2] Mobile Threat Landscape, 2017, Accessed from https://www.trendmicro.com/vinfo/us/security/research-and-analysis/threat-reports/roundup/2017-mobile-threat-landscape
[3] Introduction to Database Security Issues Types of Security Database, Accessed from http://www.academia.edu/6866589/Introduction_to_Database_Security_Issues_Types_of_Security_Database
[4] N. Davis, Secure Software Development Life Cycle Processes. Software Engineering Institute, 2013.
[5] Mobile Security, Accessed from https://www.owasp.org/index.php/OWASP_Mobile_Security_Project
[6] A. Fehnker, R. Huuck, W. Rödiger, Model checking dataflow for malicious input, Proceeding of the Workshop on Embedded Systems Security, Article No. 4, Taipei, Taiwan, October 2011.