ANDROGRAPHIC: A WEB-BASED TOOL FOR VISUALIZING AND ANALYZING ANDROID FILES

A REPORT

SUBMITTED TO THE DEPARTMENT OF COMPUTER

SCIENCE

AND THE COMMITTEE ON GRADUATE STUDIES

OF SAN FRANCISCO STATE UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

SANJAY MIRANI

MAY 2022

Abstract

As the number of Android phones grows, so does the number of mobile malware programs that execute malicious activities such as sending messages via SMS, reading users' contact information, and causing harm to the user by abusing the user's confidential data kept on the phone. Despite a large number of proposed malware analysis systems, comprehensive and effective malware analysis solutions are few and frequently fleeting. Security testing, which is regarded as one of the most important parts of the development life cycle, is either neglected or given insufficient attention; hence, security testing automation is required. One of the security tests that seek to locate potential security leaks is vulnerability detection. Fixing the breaches protects users of mobile devices such as smartphones and tablets from attacks. This paper focuses on building that tool for the users and providing them with the information they need to decide on those leaks and safeguard their interests by using or not using the application. Dividing the android code analysis techniques into two parts namely: - Static Analysis and Dynamic Analysis. The static analysis makes use of features that are extracted without executing code, whereas dynamic analysis makes use of features that are extracted during code execution (or emulation). As part of the new project entry, users will be able to upload applications to the platform for static analysis and dynamic analysis using MOBSF, a penetration testing framework, as well as for analysis of suspicious behavior within the application.

Acknowledgment

I would like to thank Prof. Abeer Aljarrah for her continuous encouragement and support over the past year. She has been an excellent project advisor and mentor. I owe a debt of gratitude for her unwavering support and for remaining flexible during these trying times. Over the last year, she has personally aided me in conquering obstacles and has been quite sensitive with me.

I also wish to thank Prof. Shah Rukh Humayoun for accepting my request to join the Defense Committee.

My sincere thanks go to all my colleagues, friends at university, and family who were constant sources of support during my master's program, always present and guiding me whenever needed.

Table of Content

List of Figures

Chapter 1: Introduction

With an 84.7 percent market share, Android is the most popular smartphone operating system today [1]. Unlike Apple's iOS, which is confined to the programs accessible in the Apple App Store, Android users are not restricted to the official Google Play Store. Instead, customers can download from a variety of sources, including third-party app stores, torrents, and direct downloads. Naturally, this freedom attracts the attention of malware producers, who try to trick users into running dangerous code by repackaging it with premium or popular apps. Although there have been reports of drive-by downloads for Android [2], user-installed malware remains the most common infection vector.

While recent research has indicated that alternative markets might contain up to 8% of harmful apps [3,4], the official Google Play Store is not immune to malware [5]–[7]. The Google Play Store is an appealing target due to the secrecy of developer accounts, the inexpensive sign-up fee of $25 USD, and its popularity among consumers, especially when considering that developers who are detected uploading malware are banned for life.

Ultimately, the end-user might consider a variety of sources while selecting whether or not to install an app:

• App ratings from other users - Ratings and comments provided by the users on the app stores like the play store and apple store based on their likes and dislikes of the application

• Permissions required by the app - Permissions regulate what an app can and cannot do. Granting Permission to the app allows it to utilize the functionality.

• Antivirus (AV) scanner results - There are several antivirus programs available that check the application and offer users scan findings indicating if the app is harmful or not.

• Google's app verification service results - This utility can keep you from installing potentially harmful apps from Google Play and other sources. It also constantly analyzes your smartphone for malicious applications.

In addition to the previous **"ANDROGRAPHIC"** project, which introduced a data visualization platform for analyzing Android datasets for its users, we added static and dynamic analyses of applications to take it to the next level and provide users with all the information necessary to determine whether an application belongs to the benign class or the malicious class by using a penetration testing framework known as **"MOBSF"**.

Chapter 2: Related Work

Most of the work has been done on comparing previous papers and applications providing different ways of classifying applications into benign and malicious. The related study can be divided into two categories according to the technique it employs: (1) static analysis techniques and (2) dynamic analysis techniques.

2.1 Static Analysis

Android malware detection using static analysis seeks to identify an application as harmful or benign based on features derived from the application's source code. The techniques presented in [1, 2, 3] construct features from the permissions requested by the apps, taking advantage of the fact that malware frequently requests dangerous/unneeded permissions. This method, however, may result in false positives because benign apps may also request dangerous permissions [1]. Additionally, since Android 6.0, the permission paradigm allows users to provide permissions only when they are required, which means that some potentially harmful permissions may never be granted. Static analysis has a number of limitations. Static analysis cannot detect malware that runs code sent after the application has started. API calls are used by other tools. DroidAPIMiner [6] classifies malware using API calls that are more commonly utilized by malware. However, due to changes in the Android API and malware development, the system must be retrained on a regular basis when new APIs are published and new forms of malware

are generated. RiskRanker [7] identifies high- and medium-risk apps based on a set of criteria, including the existence of exploit code, the usage of functionality that might cost the user money without her knowledge, and the dynamic loading of encrypted code contained in the app, and the leaking of sensitive data. During static analysis, DroidAPIMiner [8] and Drebin [9] categorize apps based on traits learned from a variety of benign and dangerous apps. However, unless they are supplemented with some type of dynamic analysis, none of these techniques can evaluate code that is obfuscated or loaded dynamically at runtime, which is a common characteristic of applications as revealed by recent large-scale research [10]. As a result, monitoring a program during run-time is critical for detecting the greatest number of rogue applications.

2.2 Dynamic Analysis

Dynamic analysis is a technique for collecting information about an application while it is running. Data collection may include battery usage, memory utilization, CPU usage, and the number of system calls to the underlying Linux kernel, to name a few examples. DroidTrace [4] monitors chosen system calls using ptrace (a system call commonly used by debuggers to control processes), allowing it to run dynamic payloads and identify their behavior as, for example, file access, network connection, inter-process communication, or privilege escalation. Canfora et al. [5] extract features from the sequence of system calls by running apps on a virtual machine, whereas Lageman et al. [11] simulate an app's behavior while running on a virtual machine using both systems calls and logcat logs.

Inspeckage [13], which also supports network traffic analysis and information collecting, is one of the tools to compare for dynamic analysis. Requested permissions, app permissions, shared libraries, testing whether the program is debuggable, and exported components are all examples of information collection.

After analyzing and comparing all the tools mentioned, all of them had their pros and cons but MobSf [6] stood out to be the best option to use. The main reasons are: -

- Mobsf combines static and dynamic analysis capabilities on a single platform.
- Its API endpoints for accessing such functions make integration into our project straightforward.
- MobSF allows you to do static analysis on both Android and iOS, as well as dynamic analysis solely on Android (currently).
- Provides an API for downloading the report as a PDF for future use.

Chapter 3: Terminology

Permissions - Android permissions are unique capabilities that applications must request before accessing critical information on your phone.

Activity - Everything you see in an Android app is done through an activity. For example, if you click on the Facebook app icon on your phone, it will display a window with your feed, but the system has initiated a Launcher activity whose job it is to display data in a specific format and also provide you with options to move to other activities such as viewing notifications or your messages.

Exported Activity - An exported activity is one that may be accessed by other components or applications. Let's take an example: suppose you're on your home screen and want to run the Twitter app; you'll click on the app icon to begin its launcher activity, but technically you were already in one activity (the home page) when you launched another (launching the Twitter app). If the Twitter app's launcher activity is not exported (not public), no one will be able to summon it, and the app will not start. That is why at least one activity must be exported so that other Android components can start it.

TLS/SSL Security test - The TLS/SSL Security test assists you in determining the security of your application's network connections. These tests are only applicable to programs that make network connections using the HTTP protocol.

Chapter 4: Architecture

This is how Mobsf is integrated and used on our platform: -

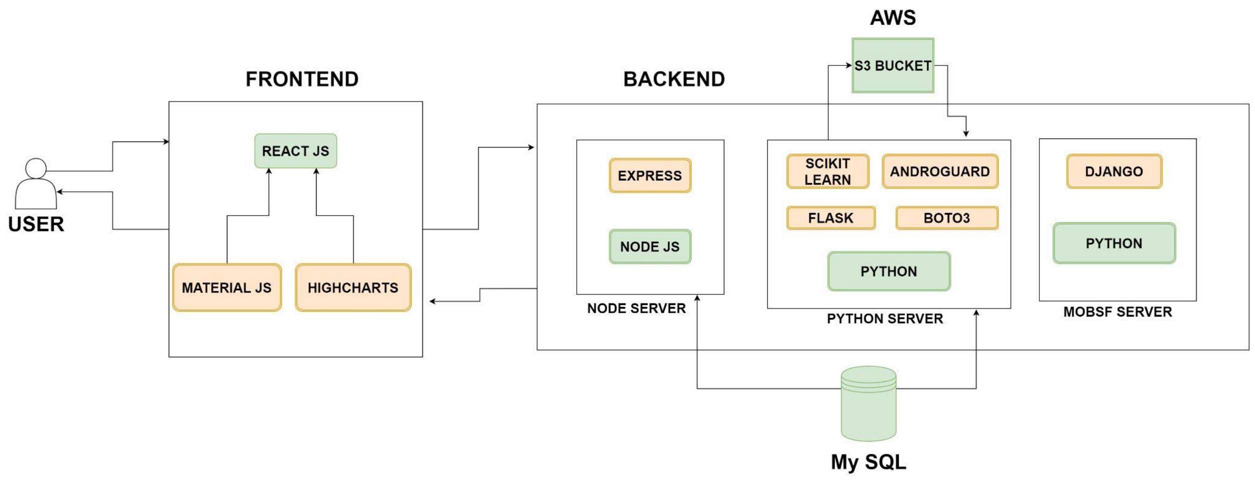


FIGURE 4.1: Architecture

AndroGraphic is a web-based utility hosted on an AWS EC2 instance, with the database running on an AWS RDS instance. React, along with a few accompanying libraries, is the framework used for front-end development. Material UI is used to generate all of the react components. We utilized Highcharts JS to visualize charts because data visualization is a major capability of AndroGraphic. Additionally, the new add-ons include a file upload page for application uploading, a page for static analysis findings, and a page for dynamic analysis results.

All incoming REST services are handled by the backend server. Express and Node JS are used to route various UI requests to appropriate REST controllers. The node server's job is to communicate with the user interface and offer relevant results by communicating with the MySQL database. A special URL pattern is used to match the request to redirect to the Python REST controller. The Flask framework is used to offer REST services on a Python server where all machine learning and Mobsf framework's API operations are done. Mobsf server is a package that is deployed on the same EC2 instance and is made up of Python and the Django framework to handle all incoming requests and reply accordingly.

Chapter 5: Implementation

The Mobile Security Framework is the tool used (MobSF). The MobSF is an automated tool for assessing the security and malware of Android applications. This tool is effective for static and dynamic application analysis. MobSF can analyze mobile applications in a variety of formats, including APK, IPA, and APPX. This framework's dynamic analysis is utilized to execute the runtime security assessment, which is also interactive. MobSF works in Mac, Linux, and Windows systems. Because MobSF is installed in the Kali Linux workstation, which is a Linux platform, the requirements for Mobile Security are:

1. Git installation - `sudo apt-get install git`
2. Version 3 of Python - `sudo apt-get install python3.8`
3. Version 8 or higher of the Java Development Kit - `sudo apt-get install openjdk-8-jdk`
4. Wkhtmltopdf - It is an open-source command-line shell program that allows users to convert any HTML (Web Page) to a PDF document. (`sudo apt install wkhtmltopdf`)

The latest addition to the project is static and dynamic application analysis, which can ultimately assist the user to understand the android application and determine how to utilize it in the future. Static and dynamic analysis are the two most popular methods of code security checks. To use those, the user would need to upload the application by clicking on the file upload tab on the left. This is what the file upload page looks like: -

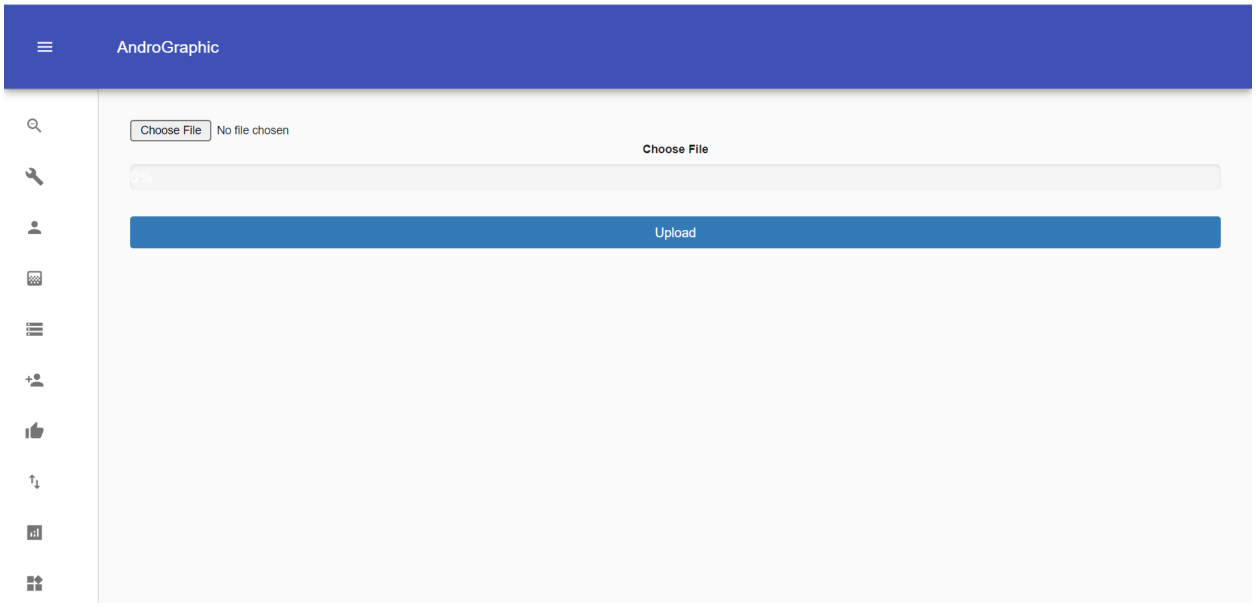


FIGURE 5.1: File Upload Page

After uploading the program, it is sent to the Python backend and stored in an Amazon S3 bucket for future usage. The purpose is to collect applications with harmful code that can later be contributed to the androzoo dataset. At the same time, the application is transferred to the Mobsf framework, where static analysis is performed and information such as application size, number of trackers and their details, permission analysis, quark analysis, and much more are acquired.

Let's deep dive into each and every component of the static analysis page: -

Start Dynamic Analysis - This will start the dynamic analysis of the application by installing it on an android emulator running constantly.

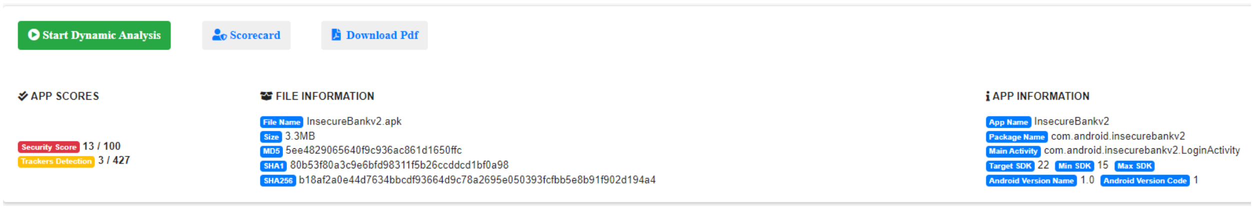
Scorecard - This will redirect the user to a page where scores related to security, trackers, and permissions are given and explained in detail.

Download PDF - This will encode all the static analysis data into a PDF file and will allow users to download it for future use.

App Score: Information about the app score, such as the icon, average CVSS, security score, and trackers detection score, is supplied.

File information: File information including the file's name, size, etc. is shown.

App Information: App information about the file is provided, including the app name, package name, primary activity, SDK android name, and code.



APP SCORES

Security Score 13 / 100
Trackers Detection 3 / 427

FILE INFORMATION

File Name InsecureBankv2.apk
Size 3.3MB
MD5 5ee4829065640f9c936ac861d1650ffc
SHA1 80b53f80a3c9e6bfd98311f5b26ccddcd1bf0a98
SHA256 b18af2a0e44d7634bbcdf93664d9c78a2695e050393fcfbb5e8b91f902d194a4

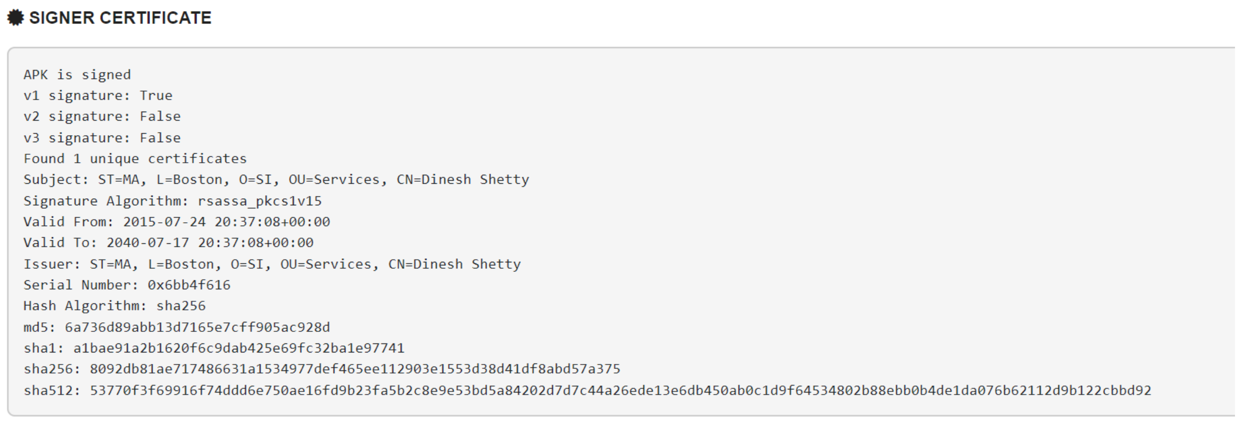
APP INFORMATION

App Name InsecureBankv2
Package Name com.android.insecurebankv2
Main Activity com.android.insecurebankv2.LoginActivity
Target SDK 22 Min SDK 15 Max SDK
Android Version Name 1.0 Android Version Code 1

FIGURE 5.2: FILE INFORMATION

5.1 Signer Certificate

This checks to see if the app came from its original source and displays certificate information. We may view the state of the signer certificate, such as warning, security, and so on, along with a description.



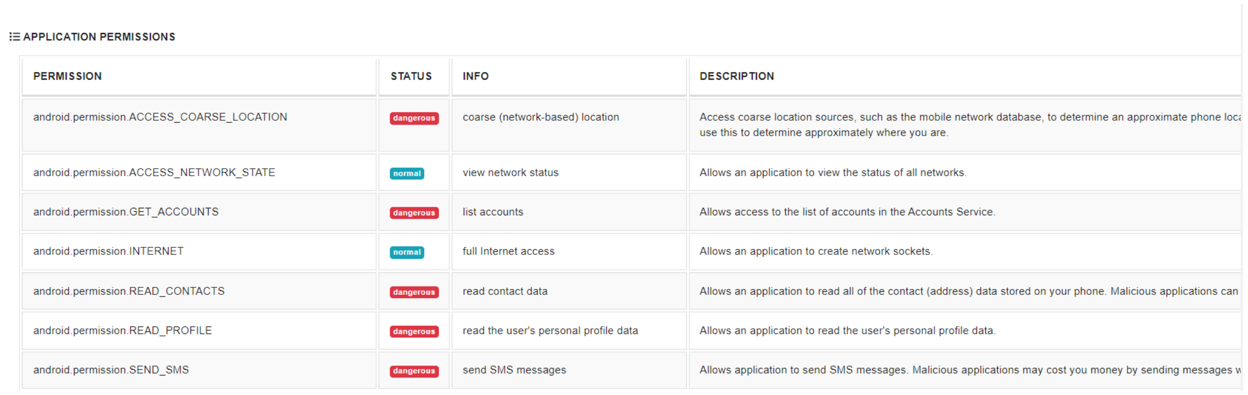
SIGNER CERTIFICATE

```
APK is signed
v1 signature: True
v2 signature: False
v3 signature: False
Found 1 unique certificates
Subject: ST=MA, L=Boston, O=SI, OU=Services, CN=Dinesh Shetty
Signature Algorithm: rsassa_pkcs1v15
Valid From: 2015-07-24 20:37:08+00:00
Valid To: 2040-07-17 20:37:08+00:00
Issuer: ST=MA, L=Boston, O=SI, OU=Services, CN=Dinesh Shetty
Serial Number: 0x6bb4f616
Hash Algorithm: sha256
md5: 6a736d89abb13d7165e7cff905ac928d
sha1: a1bae91a2b1620f6c9dab425e69fc32ba1e97741
sha256: 8092db81ae717486631a1534977def465ee112903e1553d38d41df8abd57a375
sha512: 53770f3f69916f74ddd6e750ae16fd9b23fa5b2c8e9e53bd5a84202d7d7c44a26ede13e6db450ab0c1d9f64534802b88ebb0b4de1da076b62112d9b122cbbd92
```

FIGURE 5.3: SIGNER CERTIFICATE

5.2 Application Permissions

The table provides all of the permissions that the program uses. If the user is not an Android expert, this table describes how to use the permission and also offers status according to the permissions protection level provided in the Android guidelines, such as how if the permission is dangerous, the application must expressly request the user's permission.



APPLICATION PERMISSIONS

PERMISSION	STATUS	INFO	DESCRIPTION
android.permission.ACCESS_COARSE_LOCATION	dangerous	coarse (network-based) location	Access coarse location sources, such as the mobile network database, to determine an approximate phone loca use this to determine approximately where you are.
android.permission.ACCESS_NETWORK_STATE	normal	view network status	Allows an application to view the status of all networks.
android.permission.GET_ACCOUNTS	dangerous	list accounts	Allows access to the list of accounts in the Accounts Service.
android.permission.INTERNET	normal	full Internet access	Allows an application to create network sockets.
android.permission.READ_CONTACTS	dangerous	read contact data	Allows an application to read all of the contact (address) data stored on your phone. Malicious applications can
android.permission.READ_PROFILE	dangerous	read the user's personal profile data	Allows an application to read the user's personal profile data.
android.permission.SEND_SMS	dangerous	send SMS messages	Allows application to send SMS messages. Malicious applications may cost you money by sending messages w

FIGURE 5.4: APPLICATION PERMISSION

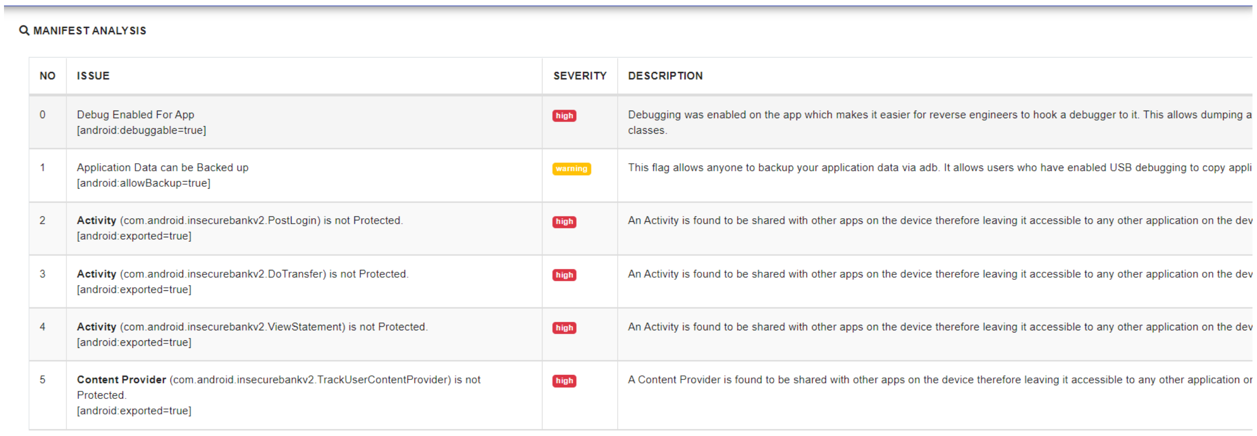
5.3 Manifest Analysis

The Android Manifest file provides system-required data including application configuration information, permissions, and app components. The Manifest Analysis table contains the information indicated above, as well as insights on Android flags such as -

BACKUP MODE ENABLED: The name usually refers to the automatic activation of backup data. The backup mode provides a backup option because the attacker can also take a backup of your data, steal important information from and use your program, leaving it susceptible.

DEBUG MODE: The debug mode determines whether or not the apps can be debugged. Furthermore, with debug enabled, the attacker has access to a wealth of information. It is a critical vulnerability that allows anyone to grab important application information.

EXPORTED TAG: The exported attribute in each manifest file component declaration specifies whether the component is private or public. If the component is public, it can be exported to other apps. However, if the component is private, other apps cannot access it, therefore this type of export option should be examined for all of those components. Otherwise, sensitive data from any component could be accessed by a third-party program.



MANIFEST ANALYSIS

NO	ISSUE	SEVERITY	DESCRIPTION
0	Debug Enabled For App [android:debuggable=true]	high	Debugging was enabled on the app which makes it easier for reverse engineers to hook a debugger to it. This allows dumping a classes.
1	Application Data can be Backed up [android:allowBackup=true]	warning	This flag allows anyone to backup your application data via adb. It allows users who have enabled USB debugging to copy appli
2	**Activity** (com.android.insecurebankv2.PostLogin) is not Protected. [android:exported=true]	high	An Activity is found to be shared with other apps on the device therefore leaving it accessible to any other application on the dev
3	**Activity** (com.android.insecurebankv2.DoTransfer) is not Protected. [android:exported=true]	high	An Activity is found to be shared with other apps on the device therefore leaving it accessible to any other application on the dev
4	**Activity** (com.android.insecurebankv2.ViewStatement) is not Protected. [android:exported=true]	high	An Activity is found to be shared with other apps on the device therefore leaving it accessible to any other application on the dev
5	**Content Provider** (com.android.insecurebankv2.TrackUserContentProvider) is not Protected. [android:exported=true]	high	A Content Provider is found to be shared with other apps on the device therefore leaving it accessible to any other application or

FIGURE 5.5: MANIFEST ANALYSIS

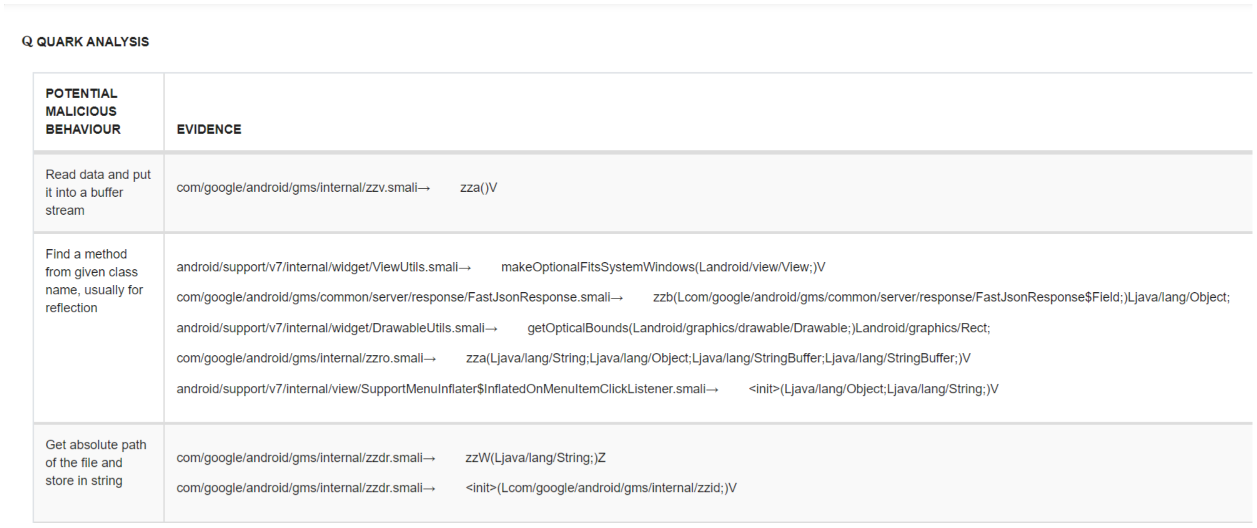
5.4 Quark Analysis

Quark Engine is free and open-source software that automates the investigation of dubious Android apps. To accomplish this, it employs a bespoke Dalvik Bytecode Loader and a one-of-a-kind scoring system that detects malicious behavior and calculates threat levels in seconds.

There are five processes to determine whether the harmful activity is being carried out.

1. Permission is being sought.
2. Call to a native API.
3. A specific mix of native API.
4. Calling the native API sequence.
5. APIs that deal with the same register.

The table summarizes the processes stated above, divided into two columns: - Potential Malicious Behaviour, which alerts the user to any possibly malicious behavior carried out by the application and corresponds to the application's Dalvik code for the same.



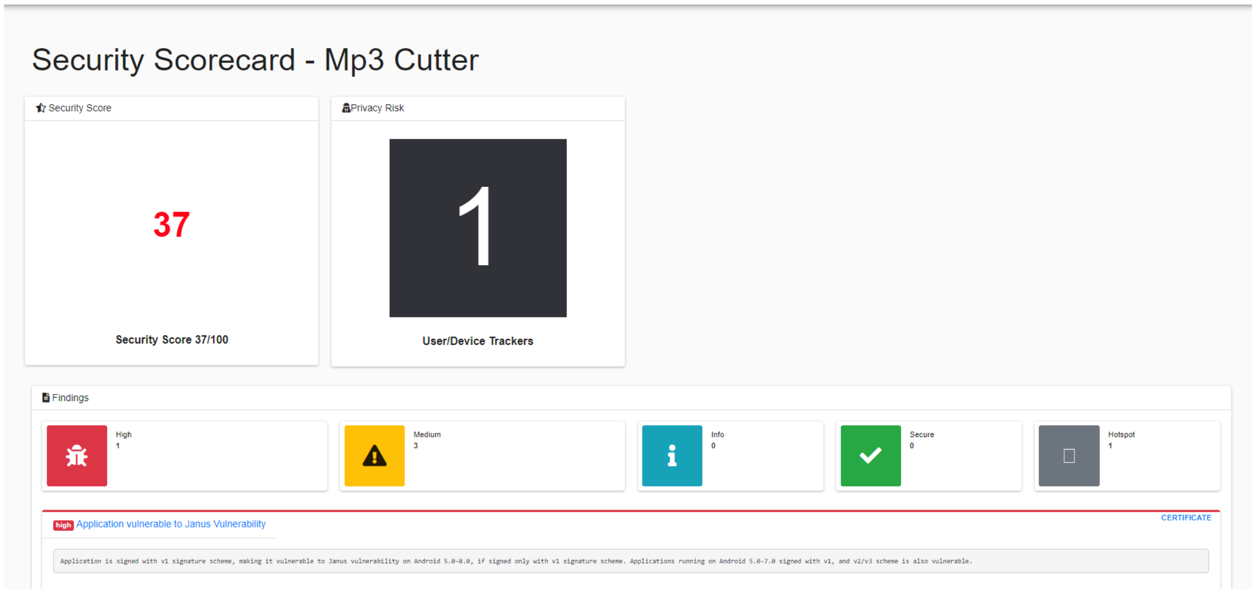
Q QUARK ANALYSIS

POTENTIAL MALICIOUS BEHAVIOUR	EVIDENCE
Read data and put it into a buffer stream	com/google/android/gms/internal/zzv.smali→ zza()V
Find a method from given class name, usually for reflection	android/support/v7/internal/widget/ViewUtils.smali→ makeOptionalFitsSystemWindows(Landroid/view/View;)V com/google/android/gms/common/server/response/FastJsonResponse.smali→ zzb(Lcom/google/android/gms/common/server/response/FastJsonResponse$Field;)Ljava/lang/Object; android/support/v7/internal/widget/DrawableUtils.smali→ getOpticalBounds(Landroid/graphics/drawable/Drawable;)Landroid/graphics/Rect; com/google/android/gms/internal/zzro.smali→ zza(Ljava/lang/String;Ljava/lang/Object;Ljava/lang/StringBuffer;Ljava/lang/StringBuffer;)V android/support/v7/internal/view/SupportMenuInflater$InflatedOnMenuItemClickListener.smali→ <init>(Ljava/lang/Object;Ljava/lang/String;)V
Get absolute path of the file and store in string	com/google/android/gms/internal/zzdr.smali→ zzW(Ljava/lang/String;)Z com/google/android/gms/internal/zzdr.smali→ <init>(Lcom/google/android/gms/internal/zzid;)V

FIGURE 5.6: QUARK ANALYSIS

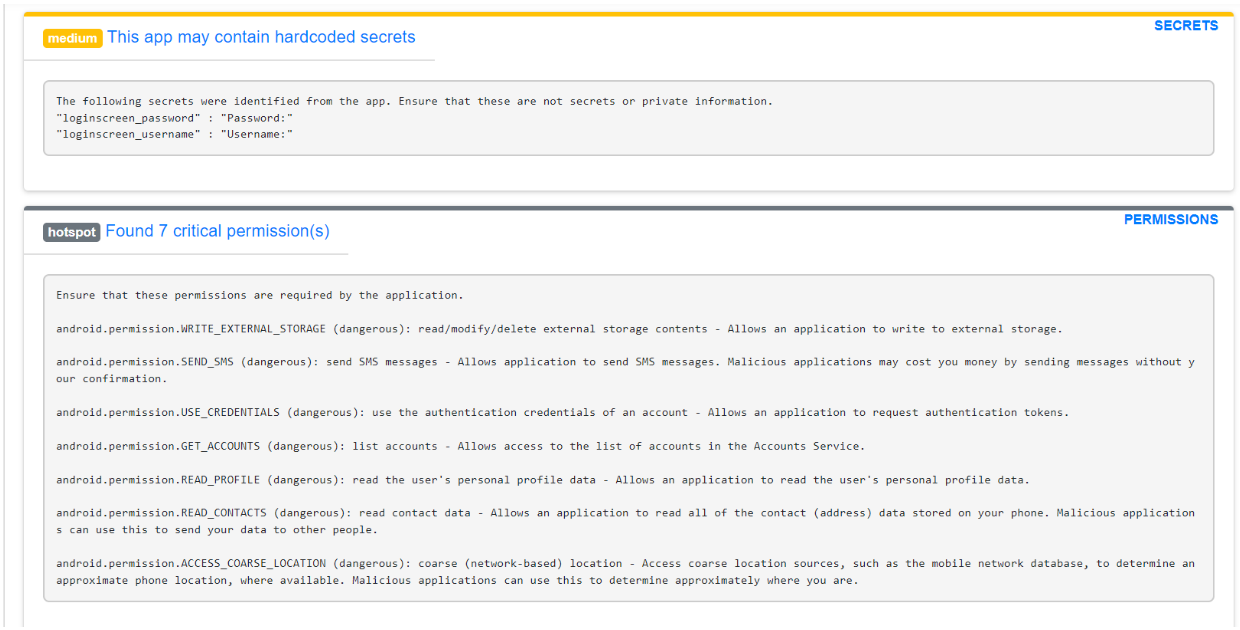
5.5 ScoreCard

The scorecard, as the name implies, provides a summary of the application's security score as assigned by MobSf, as well as insights on the permissions and trackers used in the application.



Application vulnerable to Janus Vulnerability

```
Application is signed with v1 signature scheme, making it vulnerable to Janus vulnerability on Android 5.0-8.0, if signed only with v1 signature scheme. Applications running on Android 5.0-7.0 signed with v1, and v2/v3 scheme is also vulnerable.
```



This app may contain hardcoded secrets

```
The following secrets were identified from the app. Ensure that these are not secrets or private information.
"loginscreen_password" : "Password:"
"loginscreen_username" : "Username:"
```

Found 7 critical permission(s)

```
Ensure that these permissions are required by the application.

android.permission.WRITE_EXTERNAL_STORAGE (dangerous): read/modify/delete external storage contents - Allows an application to write to external storage.

android.permission.SEND_SMS (dangerous): send SMS messages - Allows application to send SMS messages. Malicious applications may cost you money by sending messages without y
our confirmation.

android.permission.USE_CREDENTIALS (dangerous): use the authentication credentials of an account - Allows an application to request authentication tokens.

android.permission.GET_ACCOUNTS (dangerous): list accounts - Allows access to the list of accounts in the Accounts Service.

android.permission.READ_PROFILE (dangerous): read the user's personal profile data - Allows an application to read the user's personal profile data.

android.permission.READ_CONTACTS (dangerous): read contact data - Allows an application to read all of the contact (address) data stored on your phone. Malicious application
s can use this to send your data to other people.

android.permission.ACCESS_COARSE_LOCATION (dangerous): coarse (network-based) location - Access coarse location sources, such as the mobile network database, to determine an
approximate phone location, where available. Malicious applications can use this to determine approximately where you are.
```

FIGURE 5.7: SCORECARD

Now, let’s take a look at the dynamic analysis page: -

After clicking the “Start Dynamic Analysis” button, the application gets fetched from the S3 bucket and gets installed on an always running android emulator, and is been tested and monitored in the running environment. To achieve dynamic analysis we use Genymotion which is a well-known Android emulator that runs on VirtualBox. The Genymotion Virtual Devices come with a variety of widgets and sensors that may be used to imitate any real-world circumstance or scenario, including location, SMS and calls, network and baseband, and so on. You can also change the screen size and resolution to simulate any phone on the market. We are using genymotion’s Software As A Service (SAAS) version which is an EC2 instance running on AWS.

5.6 TLS/SSL Security Tester

Several TLS/SSL checks are being run on the application.

TLS Misconfiguration Test: Enable HTTPS MITM Proxy, remove the Root CA, and run the app for 25 seconds.

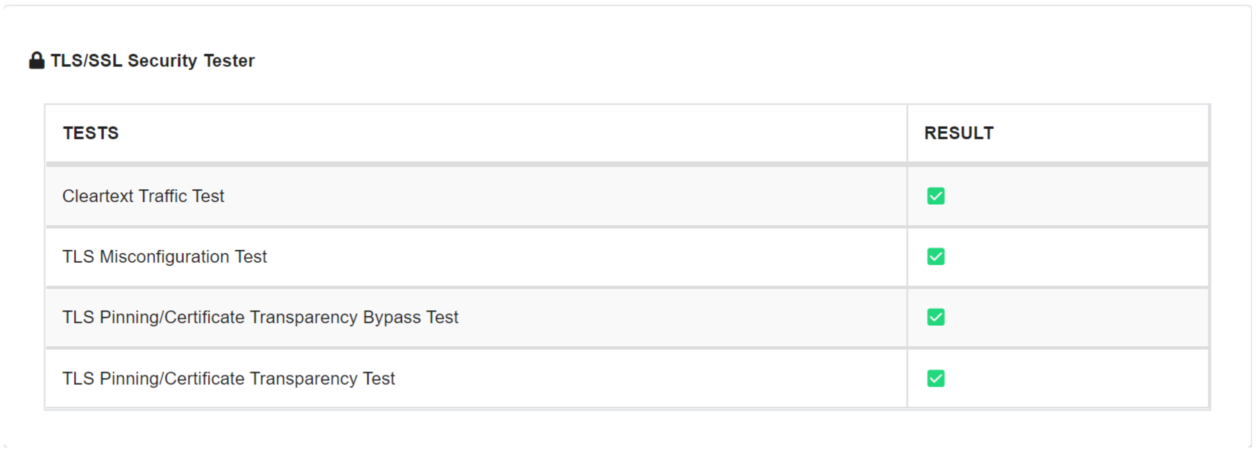
This test will identify insecure configurations that allow HTTPS connections to bypass certificate or SSL/TLS problems in WebViews. This is the same as not having TLS.

TLS Pinning/Certificate Transparency Test: Enable HTTPS MITM Proxy, install the Root CA, then run the app for 25 seconds.

This test will assess the application's TLS/SSL hardening rules and determine whether the application uses certificate or public key pinning and/or certificate transparency.

TLS Pinning/Certificate Transparency Bypass Test: Enable HTTPS MITM Proxy, Install Root CA, Bypass Certificate/Public Key Pinning or Certificate Transparency.

This test tries to go around your application's certificate or public key pinning and/or certificate transparency constraints. Most generic implementations are bypassed by MobSF.



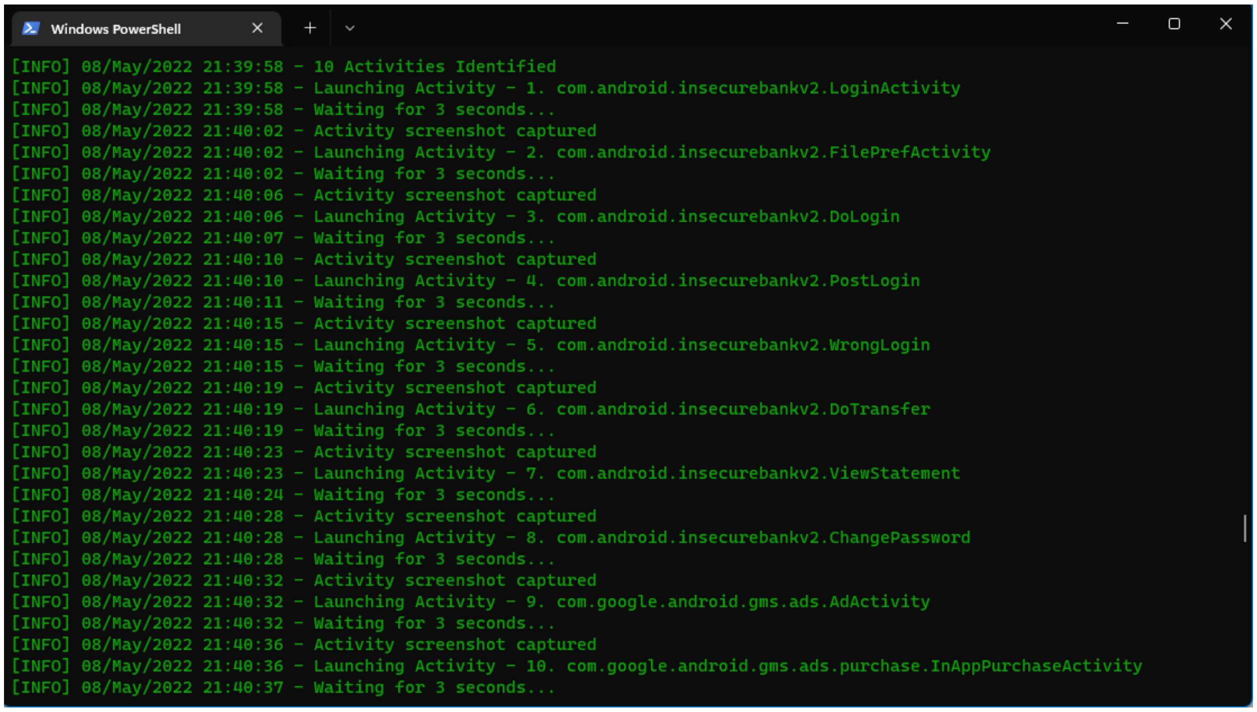
TLS/SSL Security Tester

TESTS	RESULT
Cleartext Traffic Test	✓
TLS Misconfiguration Test	✓
TLS Pinning/Certificate Transparency Bypass Test	✓
TLS Pinning/Certificate Transparency Test	✓

FIGURE 5.8: TLS/SSL SECURITY TESTER

5.7 Activity Tester

This table lists all of the activities in the program, along with a screenshot of the emulator illustrating how the application appears while the activity is active. When the activity tester is running, the terminal appears like this.

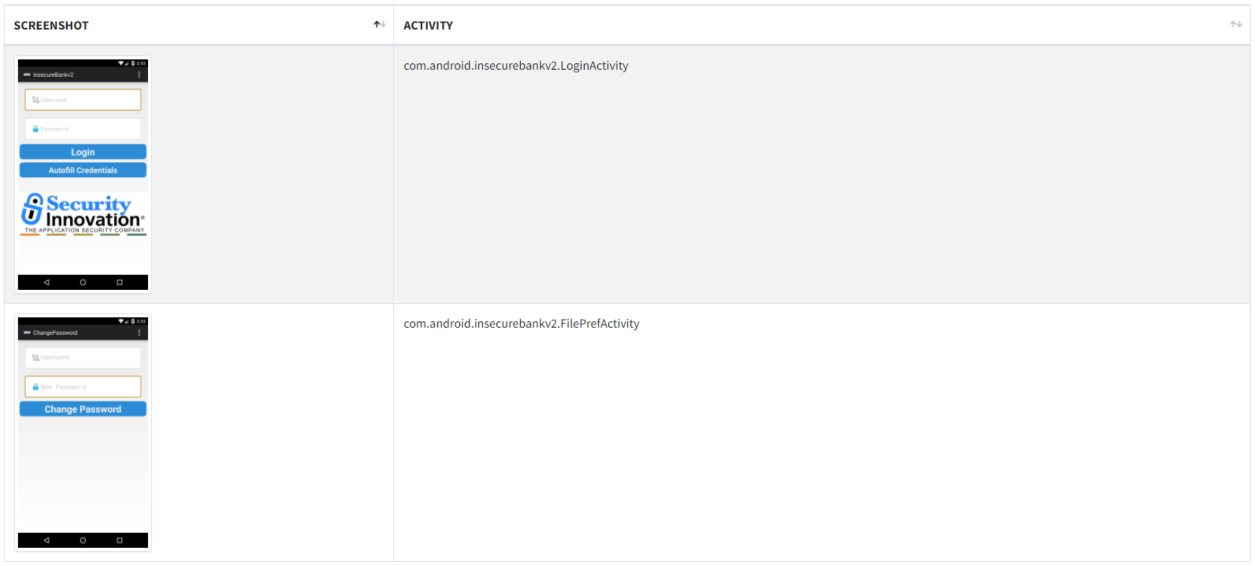


```
[INFO] 08/May/2022 21:39:58 - 10 Activities Identified
[INFO] 08/May/2022 21:39:58 - Launching Activity - 1. com.android.insecurebankv2.LoginActivity
[INFO] 08/May/2022 21:39:58 - Waiting for 3 seconds...
[INFO] 08/May/2022 21:40:02 - Activity screenshot captured
[INFO] 08/May/2022 21:40:02 - Launching Activity - 2. com.android.insecurebankv2.FilePrefActivity
[INFO] 08/May/2022 21:40:02 - Waiting for 3 seconds...
[INFO] 08/May/2022 21:40:06 - Activity screenshot captured
[INFO] 08/May/2022 21:40:06 - Launching Activity - 3. com.android.insecurebankv2.DoLogin
[INFO] 08/May/2022 21:40:07 - Waiting for 3 seconds...
[INFO] 08/May/2022 21:40:10 - Activity screenshot captured
[INFO] 08/May/2022 21:40:10 - Launching Activity - 4. com.android.insecurebankv2.PostLogin
[INFO] 08/May/2022 21:40:11 - Waiting for 3 seconds...
[INFO] 08/May/2022 21:40:15 - Activity screenshot captured
[INFO] 08/May/2022 21:40:15 - Launching Activity - 5. com.android.insecurebankv2.WrongLogin
[INFO] 08/May/2022 21:40:15 - Waiting for 3 seconds...
[INFO] 08/May/2022 21:40:19 - Activity screenshot captured
[INFO] 08/May/2022 21:40:19 - Launching Activity - 6. com.android.insecurebankv2.DoTransfer
[INFO] 08/May/2022 21:40:19 - Waiting for 3 seconds...
[INFO] 08/May/2022 21:40:23 - Activity screenshot captured
[INFO] 08/May/2022 21:40:23 - Launching Activity - 7. com.android.insecurebankv2.ViewStatement
[INFO] 08/May/2022 21:40:24 - Waiting for 3 seconds...
[INFO] 08/May/2022 21:40:28 - Activity screenshot captured
[INFO] 08/May/2022 21:40:28 - Launching Activity - 8. com.android.insecurebankv2.ChangePassword
[INFO] 08/May/2022 21:40:28 - Waiting for 3 seconds...
[INFO] 08/May/2022 21:40:32 - Activity screenshot captured
[INFO] 08/May/2022 21:40:32 - Launching Activity - 9. com.google.android.gms.ads.AdActivity
[INFO] 08/May/2022 21:40:32 - Waiting for 3 seconds...
[INFO] 08/May/2022 21:40:36 - Activity screenshot captured
[INFO] 08/May/2022 21:40:36 - Launching Activity - 10. com.google.android.gms.ads.purchase.InAppPurchaseActivity
[INFO] 08/May/2022 21:40:37 - Waiting for 3 seconds...
```

FIGURE 5.9: TERMINAL (ACTIVITY TESTER)

The sample shown illustrates that the application **"InsecureBankv2.apk"** comprises ten activities, and it will run them all one by one, taking screenshots of each one and displaying them to the user in a tabular fashion. This allows the user to see how the application and its many components will look before installing them on their devices.

An example of the screenshot of the activities is shown below: -

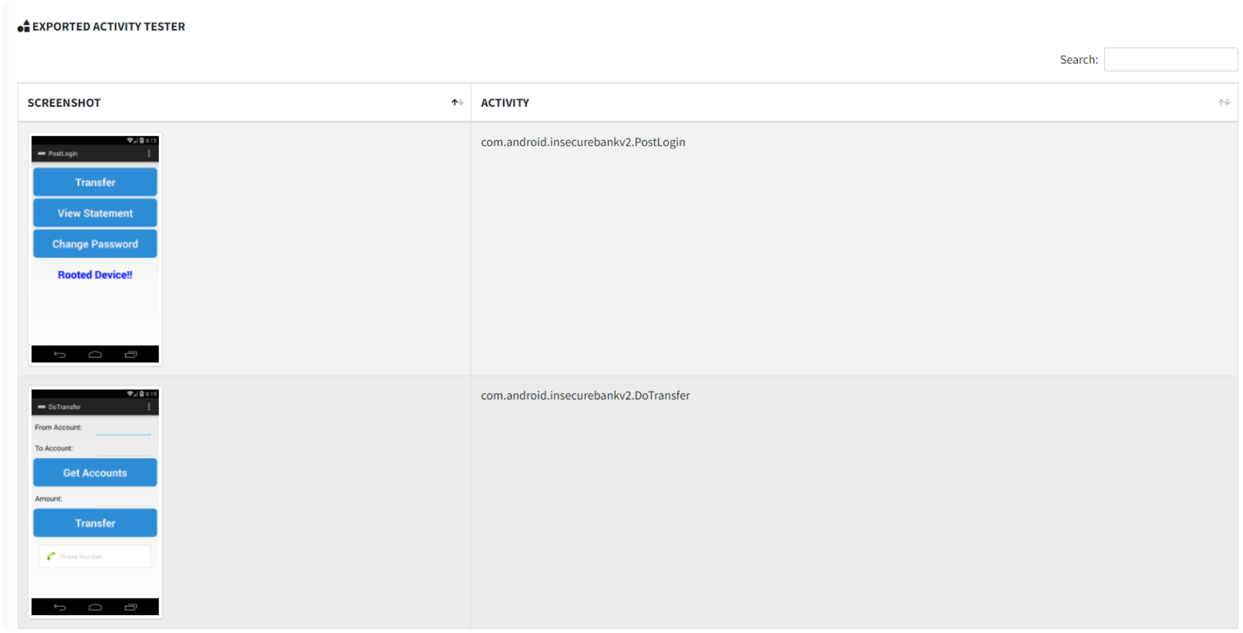


SCREENSHOT	ACTIVITY
	com.android.insecurebankv2.LoginActivity
	com.android.insecurebankv2.FilePrefActivity

FIGURE 5.10: ACTIVITY TESTER

5.8 Exported Activity Tester

The table describes the exported activities of the application along with screenshots.



EXPORTED ACTIVITY TESTER

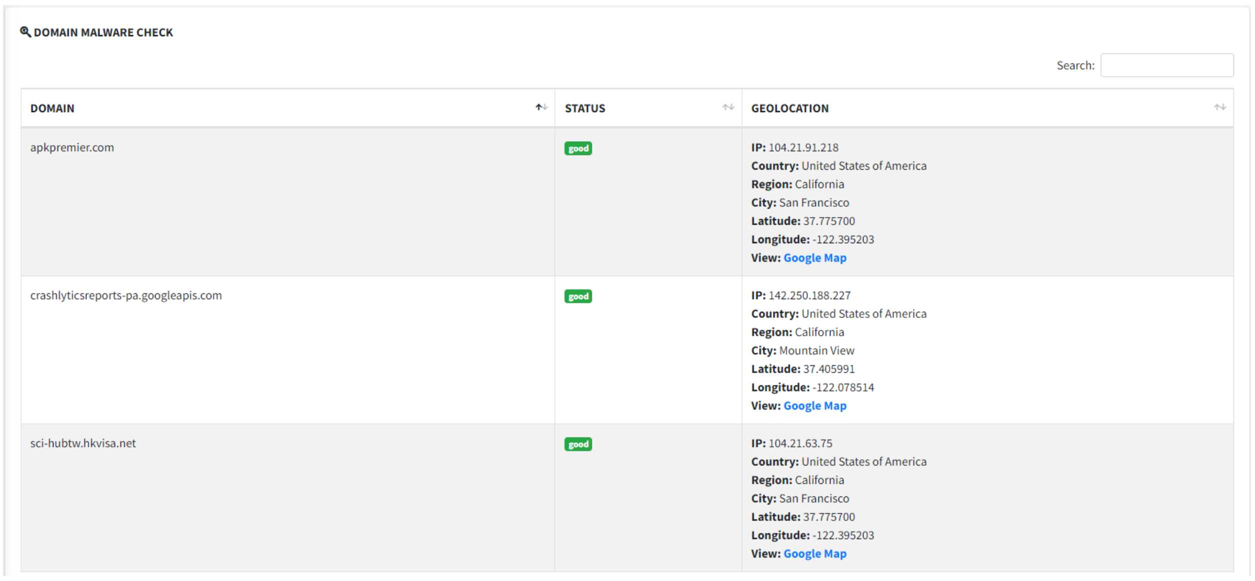
Search:

SCREENSHOT	ACTIVITY
	com.android.insecurebankv2.PostLogin
	com.android.insecurebankv2.DoTransfer

FIGURE 5.11: EXPORTED ACTIVITY TESTER

5.9 Domain Malware Check

Mobsf displays a list of URLs, IP addresses, and the files where they are stored or accessed. It Analyzes where the Android app delivers data and where it saves information.



DOMAIN MALWARE CHECK

Search:

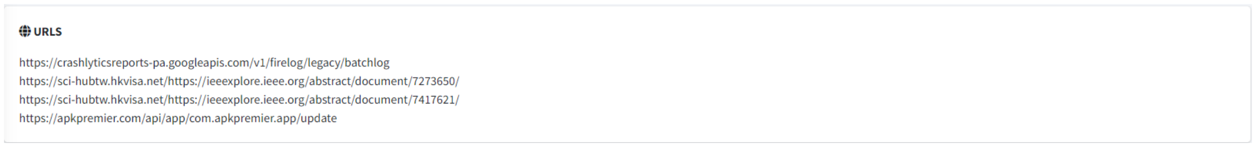
DOMAIN	STATUS	GEOLOCATION
apkpremier.com	good	**IP:** 104.21.91.218 **Country:** United States of America **Region:** California **City:** San Francisco **Latitude:** 37.775700 **Longitude:** -122.395203 **View:** Google Map
crashlyticsreports-pa.googleapis.com	good	**IP:** 142.250.188.227 **Country:** United States of America **Region:** California **City:** Mountain View **Latitude:** 37.405991 **Longitude:** -122.078514 **View:** Google Map
sci-hubtw.hkvisa.net	good	**IP:** 104.21.63.75 **Country:** United States of America **Region:** California **City:** San Francisco **Latitude:** 37.775700 **Longitude:** -122.395203 **View:** Google Map

FIGURE 5.12: DOMAIN MALWARE CHECK

5.10 URLS

Show a list of URLs, IP addresses, and the files in which they are stored or referred to.

Analyzes where the Android app delivers data and where it saves information.



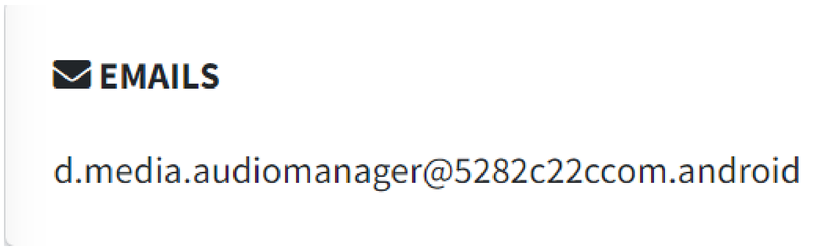
URLS

```
https://crashlyticsreports-pa.googleapis.com/v1/firelog/legacy/batchlog
https://sci-hubtw.hkvisa.net/https://ieeexplore.ieee.org/abstract/document/7273650/
https://sci-hubtw.hkvisa.net/https://ieeexplore.ieee.org/abstract/document/7417621/
https://apkpremier.com/api/app/com.apkpremier.app/update
```

FIGURE 5.13: URLS

5.11 EMAILS

The table displays the list of emails used in the application.



EMAILS

```
d.media.audiomanager@5282c22ccom.android
```

FIGURE 5.14: EMAILS

5.12 Application by Developer

We made some adjustments to the page's bubble chart tooltip to tie it to our new add-ons. If a user clicks on one of the developer's programs, they will be sent immediately to the file upload page.

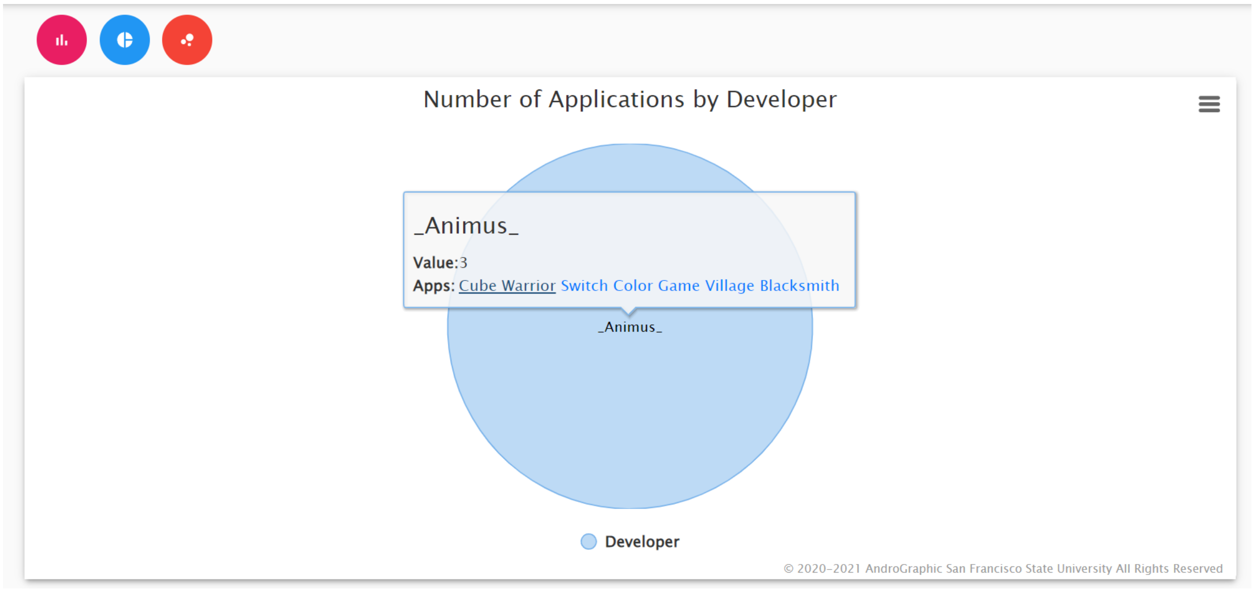

Number of Applications by Developer
Animus
Value:3
Apps: Cube Warrior Switch Color Game Village Blacksmith
Animus
Developer
© 2020-2021 AndroGraphic San Francisco State University All Rights Reserved

FIGURE 5.15: APPLICATIONS BY DEVELOPER

Chapter 6: Conclusion and Future Work

In every area, the world has become increasingly reliant on mobile applications. With the growth in demand, the rate of development of mobile applications has also grown and as a result, application testing time has decreased. Because the testing period of published apps has been shortened, these apps have become exposed to bad users and hackers who may get unauthorized access to sensitive information. We provided a practical Android vulnerability detection system add-on to the previous project throughout this study. Both developers and non-technologists can utilize the framework. Any submitted APK file is analyzed using two methods: static analysis and dynamic analysis, and an analysis report is generated. Information Leaks, Intent Crashes, Insecure Network Requests (HTTP Requests), Exported Android Components, Enabled Backup Mode, and Enabled Debug Mode were among the vulnerabilities we discovered in our app. We provide users with all the information they need to decide whether the application is harmful or not.

To make future work more user-controlled, we can provide consumers the option of selecting the number of analyses they want to apply to a certain application. Furthermore, we can provide the user greater autonomy by enabling them to choose a genymotion instance of their choosing. Genymotion supports any type of modification for the Android emulator, which can be managed using a Python script.

References

[1] W. Enck, M. Ongtang, and P. McDaniel. On Lightweight Mobile Phone Application Certification. In ACM CCS, 2009.

[2] C.-Y. Huang, Y.-T. Tsai, and C.-H. Hsu. Performance evaluation on permission-based detection for Android malware. In Advances in Intelligent Systems and Applications, 2013.

[3] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, P. G. Bringas, and G. Alvarez. Puma: Permission usage to detect malware in android. In CISIS, 2013.

[4] M. Zheng, M. Sun, and J. C. Lui. DroidTrace: a ptrace based Android dynamic analysis system with forward execution capability. In IWCMC, 2014.

[5] G. Canfora, E. Medvet, F. Mercaldo, and C. A. Visaggio. Detecting Android Malware Using Sequences of System Calls. In Workshop on Software Development Lifecycle for Mobile, 2015.

[6] Y. Aafer, W. Du, and H. Yin. DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android. In SecureComm, 2013.

[6] MobSF, “MobSF/Mobile-Security-Framework-MobSF,” GitHub. [Online]. Available: https://github.com/MobSF/Mobile-SecurityFramework-MobSF/wiki/1.-Documentation. [Accessed: 04-Dec-2019]

[7] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, “RiskRanker: Scalable and Accurate Zero-day Android Malware Detection,” in International Conference on Mobile Systems, Applications, and Services (MobiSys), 2012.

[8] Y. Aafer, W. Du, and H. Yin, “DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android,” in International Conference on Security and Privacy in Communication Networks (SecureComm), 2013.

[9] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, “Drebin: Efficient and Explainable Detection of Android Malware in Your Pocket,” in Annual Network & Distributed System Security Symposium (NDSS), 2014.

[10] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer, “Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors,” in International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS), 2014.

[11] N. Lageman, M. Lindsey and W. Glodek, "Detecting malicious Android applications from runtime behavior," MILCOM 2015 - 2015 IEEE Military Communications Conference, 2015, pp. 324-329, doi: 10.1109/MILCOM.2015.7357463.

[12] Tam, Kimberly, Salahuddin J. Khan, Aristide Fattori and Lorenzo Cavallaro. “CopperDroid: Automatic Reconstruction of Android Malware Behaviors.” *NDSS* (2015).

[13] Acpm. Inspeckage: Android Package Inspector—Dynamic Analysis with Api Hooks, Start Unexported Activities, and More. (Xposed Module). Available online: https://github.com/ac-pm/Inspeckage (accessed on 6 December 2018).