App2SecApp: Securing Android Apps through Information-flow wrapper

Submitted in fulfillment of the requirements for the degree of

Master of Technology (M.Tech)

by

Bhagyesh Patil

Roll no. **16305R003**

Supervisor:

Prof. R. K. Shyamasundar

Department of Computer Science & Engineering

Indian Institute of Technology Bombay

2019

Dissertation Approval

This project report entitled **"App2SecApp: Securing Android Apps through Information-flow wrapper"**, submitted by **Bhagyesh Patil** (Roll No. **16305R003**), is approved for the award of degree of **Master of Technology (M.Tech)** in Computer Science & Engineering.

RKShyamasundar

Prof. R. K. Shyamasundar
Dept. of CSE, IIT Bombay
Supervisor

[illegible]

Prof. G. Sivakumar
Dept. of CSE, IIT Bombay
Internal Examiner

[illegible]

Prof. Virendra Singh
Dept. of EE, IIT Bombay
External and Internal Examiner

Date: 23. June 2019

Place: IIT Bombay

Declaration of Authorship

I declare that I have written this dissertation to represent my ideas and it is completely in my own words. I have adequately cited and referenced the original sources where ever I have included other's ideas or words. I also declare that this submission adheres to all principles of academic integrity, honesty and do not misrepresents or fabricates any idea/data/fact/source. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Signature: ....Patil..............................

Bhagyesh Patil
16305R003

Date: 23. June 2019

Abstract

The number of smartphone users is increasing day by day, thus making smartphones an essential part of life. The smartphones provide many functionalities that ease user's tasks, but to provide such features, the applications residing on smartphones access user's private information (such as user's location). As the smartphone's usage has increased, the number of malicious apps has also increased. The attackers try to exploit the vulnerabilities by making malicious applications which can send the private information of the user outside the application's sandbox. This dissertation describes a technique to detect possible information leaks in apps.

There have been several approaches for securing analysis of the Android Apps. Existing approaches use either static analysis or dynamic analysis. They identify the malicious behavior of the apps by collecting logs, extracting features, and further applying machine learning approaches to classify these malicious apps. They require third-party apps for their correct functioning, such as they require a third-party tool for dismantling the 'apk' file, i.e., they need access to the 'apk' files or code-base of the apps, and also they do not offer real-time monitoring of the possible leaks during execution of the apps. The main problem persists with getting the source code of large number of applications.

We developed an wrapper which detects the sensitive data flowing outside the application's sandbox; this wrapper acts as an extension to the apps. As there is no single entry point in an Android app, it makes it challenging to analyze the flow of information just through static analysis. We use dynamic tracing technique to find out through which provided permissions of Android framework a malicious app can leak information. Our approach detects the possible leaks while doing real-time monitoring of the application, and neither require any access to the app's code, nor does it require apk analysis. For monitoring the application, we place a wrapper, an android library, called App2SecApp, around the given app with the following properties: (1) Semantic behavior of the app remains the same except for classes and APIs that are used for sending sensitive information outside the app's sandbo. (2) App2SecApp monitors the sensitive API calls and prompts the user, if there is a possible leakage from the app's sandbox through any of the API call. To evaluate our solution for its performance in terms of ease of use, efficiency and user-friendliness, we added it to Pedometer, a popular open-source Android application. The results show that our solution changed only 0.11% of the LOC of the original source code. We evaluate the user experience of our solution by a survey which shows that none of the participants felt the UI interference and were pleased as long as apps security is integrated.

Contents

List of Figures

Chapter 1

Introduction

Android framework[1] runs each application in its sandbox. If the application needs to access the data residing in the user's smartphone or want to access resources outside of its sandbox, then the application has to request permission explicitly. User's get doubtful when the application's ask for permissions without showing why and how they are using it. When a user grants permissions[2] to the application, it trusts the application developer for its private data. Malicious apps can expose the user's data over the network, through SMS, or by writing it in an external storage file.

When a person installs a new application, it asks for user's permission before accessing personal information and specific hardware features. Ideally, this knowledge is for the proper functionality of the application. For example, cab booking application will not be of any use if we do not give access to the device's GPS information to it. Once we give permission to access user's data, it can share this data through sensitive API calls outside the application's sandbox. Apps such as Google Maps[3] or Facebook[4] are continuously monitoring our location for their proper functionality.

An enormous amount of work has been done, and certain tools and techniques exist which can detect malicious Android apps up to a certain extent. Most of the existing approaches use either static analysis[5] or dynamic analysis[6], and some use both. They identify the malicious behavior of the apps by collecting logs, extracting features, and further applying machine learning approaches to classify these malicious apps.

Static Analysis

Static analysis is a technique which detects the presence of malware by analyzing the code of the application. It does not require running of an app onto emulator or device; hence, it requires less time and resource than dynamic analysis. This technique suffers from code obfuscation, i.e., the developers deliberately make the code that is difficult for humans to understand and thus can hide the malicious behavior of the apps easily. Also, the code analyzed during static analysis may not necessarily be the code that will be running during the execution of the app.

Dynamic Analysis

Dynamic analysis detects the malicious behavior of the apps while they are running on an emulator or device. It should cover every possible run-time behavior of the app and should include every execution path possible. It is immune to the code obfuscation as we detect the malicious behavior on the actual execution path. It is often time-consuming and less scalable.

Our Approach

Our approach is a dynamic analysis technique which does not require any access to the code base of the app. Also, it does not need any dismantling of the 'apk' of the app to convert it into modified 'apk' as done by previous approaches. We developed an Android library which acts as an extension to the applications; the developers should use this library to make a secure app which gives the authority to the user to take the decisions according to his/her will. Our library act as an interceptor, whenever a sensitive API call is called it gets intercepted by our library and desired pre-operations are performed after that the actual API call is made. A message is prompt to the user if there is a possible leakage from the application's sandbox through any of the API calls, then the decision has to be taken by the user according to his/her will. We will look into it in detail in subsequent chapters.

Organization of Dissertation

The rest of the dissertation is organized as follows: Chapter 2 gives an overview of the

Android OS, Android Architecture, which provides the basic structure of different components of Android and how they interact. It also contains a summary of previous researches done on Android security and describes Security in Android, which outlines different component, ensuring android security. Chapter 3 describes our approach to detect malicious behavior of the applications. Chapter 4 provides the steps to be followed to integrate App2SecApp in the application. Chapter 5 gives details about the implementation of our approach. Chapter 6 illustrates different case studies of malicious apps and how our method successfully detects their leaky behavior. Chapter 7 evaluates our approach on an open-source android app. Chapter 8 provides conclusion and future work.

Chapter 2

Background and Related Work

.

2.1 Background

This section provides the reader with a brief overview of the Android OS. Firstly we give a high-level outline of the Android Architecture and, Android application essential components and how they interact and work together. Then we describe the permission-based access control used by Android for accessing user's private data and hardware access.

2.1.1 Android architecture

The Android operating system[7] is a stack of open-source, Linux-based software components. It is roughly divided into five sections and four main layers, as shown in the figure 2.1 [1].

1. **Linux Kernel**
   Linux Kernel manages all the drivers such as display drivers, camera drivers, memory drivers, etc. which are mainly required by the Android device during the runtime. It is responsible for memory management, power management, device management, resource access, etc. The central security purpose of the Linux Kernel is to separate apps resources from each other when multiple apps run on Android OS, e.g., A app

cannot use other app's memory space or CPU resources or read another app's data. Also, apps cannot use services like Camera, SMS, etc. without permission by Linux Kernel.

2. **Hardware Abstraction Layer (HAL)**

   Android HAL acts as middleware between hardware and software. Using HAL allows you to implement functionality without affecting or modifying the higher level system. HAL includes various library modules to provide an interface to hardware component such as Bluetooth, camera, etc. When any API makes a call to access device hardware, Android system loads that corresponding library module for that hardware.

3. **Android Runtime**

   Android runtime(ART) provides a key component called Dalvik Virtual Machine, which is a kind of Java Virtual Machine specially designed and optimized for Android. The ART is the engine that powers our applications along with the libraries and it forms the basis for the application framework. ART is written for low-memory devices to run multiple virtual machines by executing DEX files.

4. **Native Libraries**

   These are a set of Java-based libraries that are specific to Android development. These libraries include the application framework libraries that facilitate user interface building, graphical drawing, and database access. These core libraries are, in fact, essentially Java "wrappers" around a set of C/C++ based libraries. With the help of Java API framework, we can access these native libraries through apps.

5. **Java API Framework**

   Android provides many higher-level services to the application developers in the form of Java classes. These classes are reusable, interchangeable, and replaceable components and act as a building block for the android applications.

6. **Applications**

   Android provides core applications like contacts, calendars, email, etc. All the native applications or third-party applications reside at this layer.

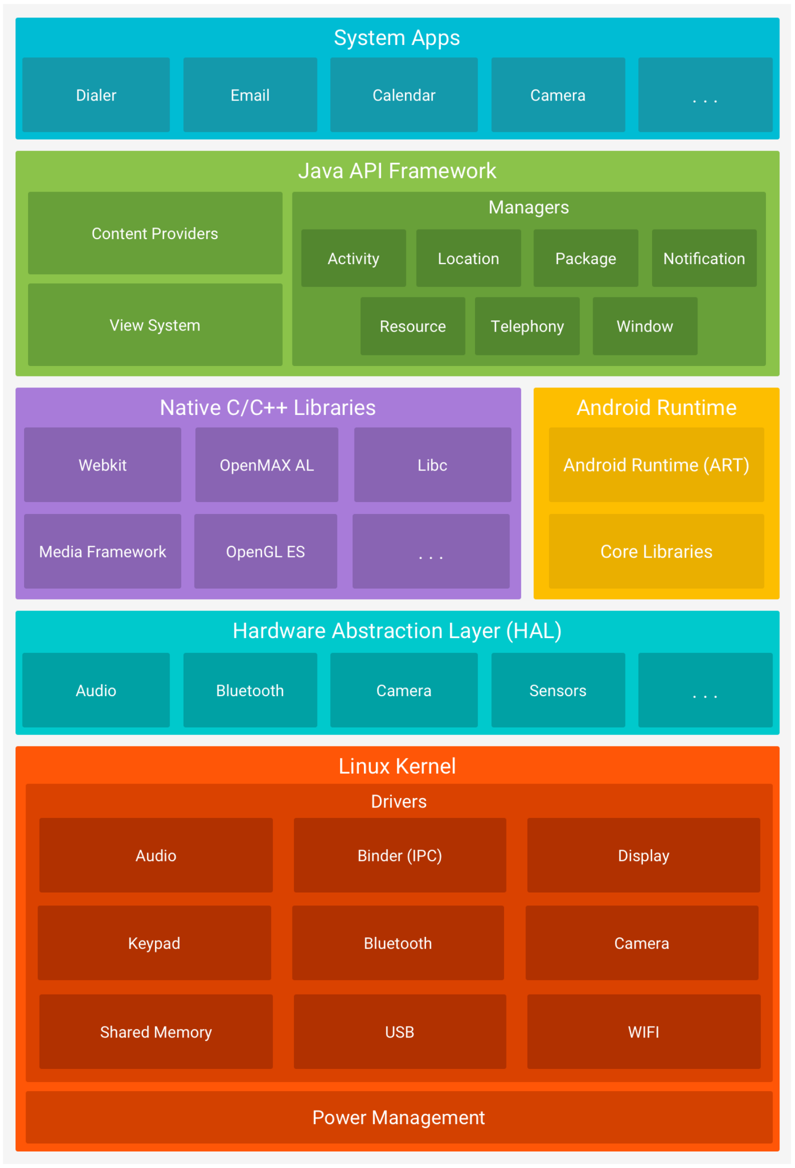


Figure 2.1: The Android software stack

2.1.2 Android App Components

We can see Android applications as a group of various components[8] interacting with each other to perform a desired task. To use a component, the app has to explicitly request that particular component in the application's manifest file. The manifest file defines the set of components and permission the app will require during its entire execution. There are four types of app components.

1. **Activity**

Its a fundamental block of building an android application. An activity represents a single screen on an app and allows the user to interact with it. The activity has its life cycle and goes through various stages depending upon the state of the activity.

2. **Service**
   A service is a component that runs in the background to perform long-running operations without a user interface. They do not require the app to be running, e.g., we can close an application while a file is getting downloaded by it.

3. **Broadcast Receiver**
   Broadcast Receivers respond to broadcast messages from other applications or from the system. e.g., whenever we turn on the device, a broadcast is sent that the device is up to the registered processes.

4. **Content Provider**
   Content provider provides the applications with the data; it acts as a mediator application requesting data and a data provider such as a database. It is useful when data is to be shared across applications.

2.1.3 Permission System

The main aim of the permission system[2] is to protect the user's privacy. Toto access sensitive user data (such as contacts and SMS), as well as certain system features (such as camera and internet), Android apps must request permissions. All the permissions needed by the application must be mentioned in the manifest file of the app. The user has to give permissions to the app for the smooth functioning of it; no permissions are granted to the app in advance.

Requesting Permissions

Android system uses the sandboxing technique where each app runs in its sandbox and cannot access resources outside of it. The application has to explicitly request for resources and data residing outside of its sandbox if it wants to access it. All the permissions required for accessing the resources must be declared in the manifest file. Figure 2.2[10] shows access to sensitive data available through protected APIs. Depending upon the sensitivity of the permission, the Android OS may grant it automatically or may prompt the user or reject it. System permissions are divided into several protection levels. Three protection levels affect third-party apps:

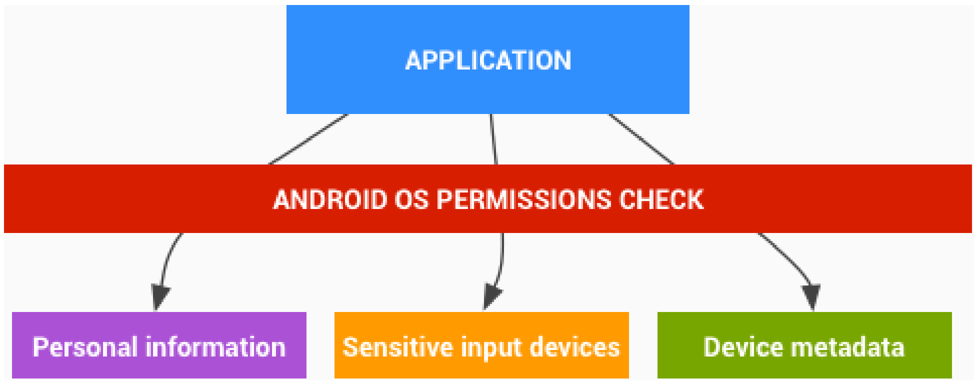


Figure 2.2: Access to sensitive data through protected APIs

- **Normal Permissions**

  Normal permissions are granted automatically by the system. They cover areas where your app needs to access data or resources outside the app's sandbox, but where there's minimal chance of leaking the user's private information or very little risk to the operation of other apps. e.g., permission to set the time zone is a normal permission. These permissions are also needed to be mentioned in the manifest file, but the user is not prompt to grant these permissions.

- **Dangerous Permissions**

  They cover areas where your app needs to access the user's private data or resources or can affect the functioning of other's apps. These permissions are mentioned in the manifest file of the app and have to granted explicitly by the user. e.g., permission to access the contacts residing on the device is a dangerous permission.

- **Signature Permissions**

  Signature permissions are automatically granted when another app which exists on the same system already has the requested permission and is signed by the same developer. The system automatically grants these permissions at the install time.

2.1.4 Security in Android

Android provides some security features, e.g. it provide secure inter-process communication facility so that process should not run in isolation with other processes.[9, 10]

- **Linux Security**

  Android's foundation is Linux OS. Linux provide security features such as process isolation.

- **Application Sandboxing**

  Applications are not allowed to access resources outside the application's sandbox.

- **System Partition**

  Partitions such as Android Kernel and operating system core libraries are read-only and cannot be altered.

- **Cryptography**

  Android provides a set of cryptographic APIs that can be used by application developers.

- **Cost-Sensitive APIs**

  Cost Sensitive API (like Telephony, SMS/MMS etc) are placed in protected APIs set which is controlled by Android System.

- **Sensitive Data Input Devices**

  Sensitive data input devices include camera, microphone, or GPS. To access them user must explicitly provide permission to the application.

Even after the features provided by Android, the security leak is possible because of the malicious programming where the developers can touch the sensitive information of the user and can send it outside the app's sandbox revealing it to the outside world. It is possible if required permissions are acquired by the application or are given by the user unknowingly.

The security can be inferred for the applications in the following ways:

1. Using the language source of the app, one can initially assign 'a security label' for the sensitive data and can monitor the flow of such sensitive data and assign appropriate label to the other objects touched by such sensitive data. If a high sensitive data tries to go outside the apps sandbox, then it might be a security leak.

2. If the language source is not available, then dynamic analysis of the app can be done where the the execution traces of the app are defined and if a app tries to violate the traces then there might be a possible security leak.

3. We can monitor the permissions affecting the application and trace the data being affected by these permissions i.e. the data coming to and fro of the apps. If a sensitive data is touched by the app and it is trying to send it outside the apps sandbox, then there might be a possible leak.

2.2 Related Work

2.2.1 DroidDolphin

DroidDolphin[11] is a dynamic malware analysis framework which makes use of the technologies of GUI-based testing, big data analysis, and machine learning to detect malicious Android apps. It collects runtime logs of Android apps and applies machine learning techniques to check whether the app is malicious or not. It uses both dynamic and static analysis. The DroidDolphin framework contains four phases: Preprocessing, Emulation and Testing, Feature extraction, and Machine Learning. In preprocessing, it monitors specific API calls that may be called by malware; it uses APIMonitor for this purpose. In the emulation and testing phase, it uses APE[12], which is an automatic Android malware testing tool developed to simulate GUI-based events and traverse application's code paths, it triggers GUI events by identifying the buttons and filling forms on the touch screen. Logs are gathered from this phase for further analysis. Using the logged data provided by the previous two phases, the features for each app are extracted in the next phase. And in the last phase SVM machine learning algorithm is used to build a malware prediction model. The success of such a dynamic analysis approach depends on two issues:

- Whether collected logs are sufficient and useful and
- How well the machine learning techniques are applied.

Anti-emulation techniques are used to hide apps from dynamic analysis with the emulator. The code coverage done by this tool is less, and the time consumed is more.

2.2.2 Dynaldroid

Dynaldroid[13] automates the app execution on an emulator to facilitate dynamic analysis of the app. The information collected during the dynamic analysis of malware samples generates test cases that can indicate the occurrence of malicious activities. The network activity carried out by the application is also analyzed by the Dynaldroid. Dynaldroid runs in 3 phases, namely Preparation phase, Execution phase, and Analysis phase. The test case for application execution is generated in the preparation phase; various events are created for different types of UI element. Systematic event triggering follows DFS as shown in the figure 2.3[13]. Execution phase constitutes of test case execution. In the analysis phase, all the information gathered from the above two phases (system calls traces, API calls, and network activity traces) is then parsed to identify a set of predefined patterns which indicate the occurrence of malicious activity.

2.2.3 SmartDroid

SmartDroid[14] uses both static and dynamic analysis to reveal UI-based trigger conditions in Android applications. It triggers a certain behavior through automated UI interactions efficiently. It uses static analysis to extract expected activity switch paths by analyzing both Activity and Function Call Graph (Figure 2.4[14])and then uses dynamic analysis to traverse each UI elements and explore the UI interaction paths towards the sensitive APIs. Compared to other dynamic analysis tools, it takes less time in revealing the sensitive API calls.

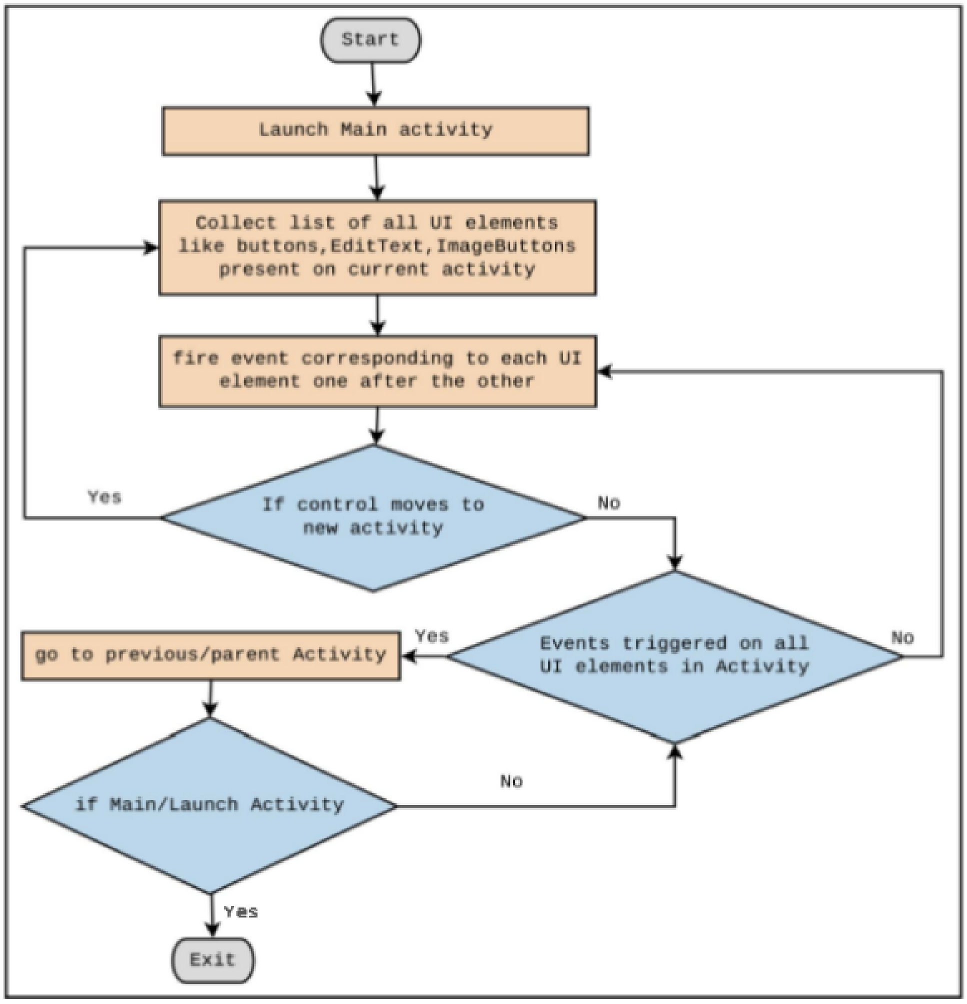

Start
Launch Main activity
Collect list of all UI elements like buttons,EditText,ImageButtons present on current activity
fire event corresponding to each UI element one after the other
If control moves to new activity
Yes
No
Events triggered on all UI elements in Activity
Yes
No
go to previous/parent Activity
if Main/Launch Activity
No
Yes
Exit

Figure 2.3: Flowchart showing automated app execution in DynalDroid

2.2.4 Dynalog

Dynalog[15] is an automated dynamic analysis framework for characterizing Android apps. Dynalog framework accepts large numbers of Android apps, launch them serially in an emulator, and log several dynamic behaviors. It provides an automated platform for mass analysis and characterization of apps that is useful for quickly identifying and isolating malicious applications. It uses existing open source tools to extract and log high-level behaviors, API calls, and critical events that can be used to explore the characteristics of an application. Some Machine learning classification method is used to classify benign and malicious app.

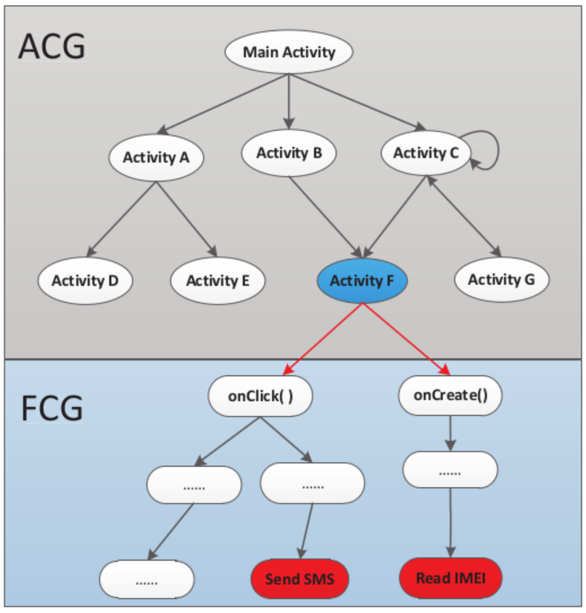


Figure 2.4: ACG and FCG

2.2.5 SecFlowDroid+

SecFlowDroid+ [16, 17] is a tool developed to point the possible leakage points in the apps. They developed a hybrid approach for it that utilizes both dynamic and static analysis approaches. First, it dynamically executes the given application in APIMonitor, which generates a sequence of protected APIs that are invoked during the manual exploration of the app. Logs are collected which are to be used for static analysis. Then, it extracts the JIMPLE code of those methods which has invoked protected APIs using SOOT-Android tool. It analyzes these JIMPLE files and tracks information flow, including implicit flows, between API invocation using RWFM model and report misuse if found.

2.2.6 Merits of App2SecApp

The literature studied in this Chapter uses various open source third-party tools to automate their processes for collecting logs of the sensitive API calls called by the application, which in fact is a crucial step in determining the malicious behavior of the apps. In contrast, App2SecApp do not require any third-party tools; the wrapper provides all the functionalities.

Existing approaches require the ‘apk’ file of the applications for their analysis, and some require code-access of the app. Getting the ‘apk’ file of an application is not an easy task as most of the users download the apps from the Google Play store. The ‘apks’ available on the web are the old ones and couldn’t be trusted. In contrast, App2SecApp do not require any type of access to the code. In fact, all the analysis and monitoring is done by the wrapper at the run-time.

The previous methods do not provide real-time monitoring of the malicious behavior of the apps while the user is using it, they do dynamic analysis by running the app in an emulator and then collects logs to determine its behavior whether it is malicious or not. In contrast, our approach provides the real-time monitoring of the app and give authority to the user, to deny an action if it is undesirable.

Chapter 3

Our Approach to Detect Malicious Behavior of the Applications

In this chapter, we describe our approach to detect the malicious behavior of the applications. We have developed an wrapper, named App2SecApp, which acts as an extension to the current applications; the developers should use this wrapper to make a secure app. It neither requires any access to 'apk' file of the application nor any access to the code files of the application. It provides real-time monitoring of the app while a user is using it in his/her device. It gives the authority to the user to take the decisions according to his/her will. Our wrapper act as an interceptor, whenever a sensitive API call is called it gets intercepted by our wrapper and desired pre-operations are performed after that the actual API call is made. A message is prompt to the user if there is a possible leakage from the application's sandbox through any of the API calls, then the decision has to be taken by the user according to his/her will.

We analyzed the architecture of the Android and its components and analyzed how an application can access the private information and resources outside of its sandbox from the internal storage and hardware. For this, the Android framework provides different API methods which can be called by the application for the desired access of resources. These API methods can be divided into two categories i.e. Source API calls and Sink API calls.

- **Source API calls:** These are the API calls that collect information from outside of apps sandbox. The information flow from outside to the inside of the app. e.g., getLatitude() and getLongitude() access location of the device and provide it to the application.

- **Sink API calls:** These are the API calls that send information to outside of the sandbox. The information flow from inside to the outside of the app. e.g., sendTextMessage() send SMS from app.

These are the sensitive API methods as the information is getting traveled to and fro apps sandbox. These sensitive API methods are contained in classes. For the proper functioning of these methods and classes, Android uses a permission-based system. If the desired permission is given by the user to the application, then the corresponding API method will work adequately achieving its desired result. If the user does not provide the permission, these API methods will not give the desired result. Android framework contains the following permissions:

- **Location:** ACCESS_COARSE_LOCATION, ACCESS_FINE_LOCATION
- **SMS:** READ_SMS, RECEIVE_SMS, SEND_SMS, RECEIVE_MMS
- **Phone:** READ_PHONE_NUMBERS, READ_PHONE_STATE, CALL_PHONE
- **Storage:** READ_EXTERNAL_STORAGE, WRITE_EXTERNAL_STORAGE
- **Camera:** CAMERA
- **Contacts:** READ_CONTACTS, WRITE_CONTACTS
- **Call Logs:** READ_CALL_LOG, WRITE_CALL_LOG
- **Calendar:** READ_CALENDAR, WRITE_CALENDAR
- **Microphone:** RECORD_AUDIO
- **Internet:** INTERNET
- **Sensor:** SENSOR

We found that the attackers can misuse these permissions, thus resulting in the malicious behavior of the app. Such apps can leak the private information of the user. e.g., If an app has location and SMS permissions, then it can send the user's location to an unknown number embedded in the app's code.

We observed that all the information flows through with the help of these sensitive API calls. So to monitor this information flowing, we came with the idea of intercepting each of these sensitive API calls through the wrapper. App2SecApp android wrapper:

- Extends the classes containing the sensitive API calls.
- Maintains a Database 'Permissions.db' which include all the permissions which have influenced the application so far.

Now, each API call executed by the application does some pre-processing before calling the actual API call. This pre-processing depends on whether the API call is source API call or a sink API call. If it is:

- **Source API call** then information is flowing into the application, i.e., your application is influenced after that API call is called. In this case, the permission corresponding to that API call is entered into the database. e.g., When getLatitude() API method gets called, the wrapper adds the 'Location' permission into the database before calling the actual API call.
- **Sink API call** then information is flowing from the application outside its sandbox, i.e., your application is sending some data to the outside environment. In this case, the user gets prompt that the following permission influences the app and now is trying to send data outside the application's sandbox through this API method.Now the user has the control to continue with the operation or to deny it.
  - If the user chooses to continue with the operation, then the actual API call is called.
  - If the user rejects, then the call is terminated in the pre-processing phase, and no actual call is made to the API method.

e.g., When an application has already accessed device's location and calls sendTextMessage() API method, then the wrapper prompts the user that the app is being influenced by the 'Location' permission and now wants to send a text message. Now the user has the control to continue with the operation or to deny it.

Figure 3.1 shows the steps that take place when a user tries to interact with a app integrated with App2SecApp wrapper:

1. User interacts with the Application.
2. If a sensitive Source or Sink API method is called then DB Module is consulted by the Extended Class Module.
3. If the API call is:
    - Sink, then the Extended Class Module prompts the User with appropriate message.
    - Source, then the next step is skipped.
4. If the User continues with the operation, the original Default Class is called corresponding to the sensitive API method invoked. Otherwise not.
5. The result is sent to the Extended Class Module.
6. The Extended Class Module sends the result to User.

Prompts will be displayed to the user whenever something is sent out of the app's sandbox, so the user may sometime get annoyed. We can try to improve the User Experience by applying Machine learning techniques which will attempt to learn the inputs given by the user for a particular execution path or trace for a given number of times. And then after learning, the model can automatically hide the prompt by using the previously learned inputs for a specific user.

This approach successfully mitigates the malicious behavior of the application; in fact, it gives the user the overall control of the app. The app user now has a clear visualization of "at what point, on which activity, or on what event his/her data is sent outside of

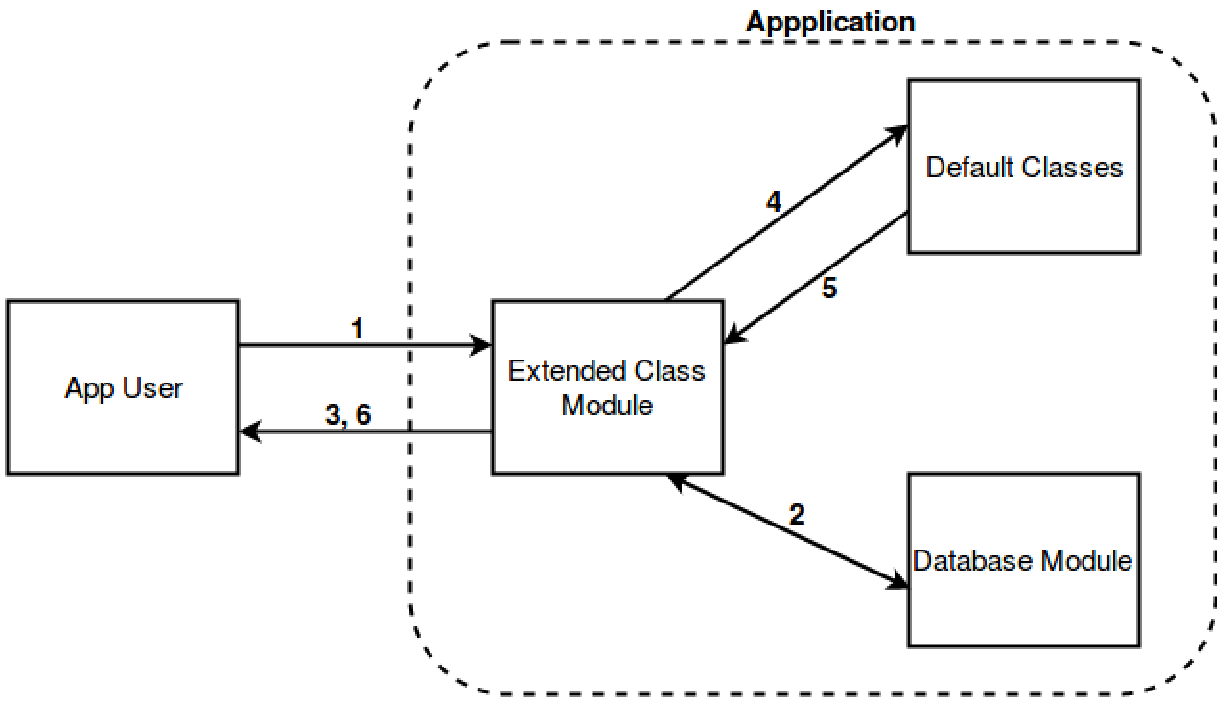


Figure 3.1: Steps that take place when a User interact with App2SecApp.

the app". If the user thinks that it is the desired behavior of the app, then he/she can continue, otherwise can deny. More importantly, it gives the real-time monitoring of the API calls when the user is using the application in the real environment. The approach can be easily be extended to determine the inter-application communication as done in [18]. We can monitor the communication done via implicit intents[19] , which are used for sending data outside the app's sandbox to another app. This monitoring will detect that the app is trying to communicate with another app and will prompt the user the same, then the user can take the necessary decision. We illustrate certain example application which use App2SecApp wrapper in Chapter 6.

Chapter 4

Realizing a New App with a Wrapper around the Old App

Our wrapper, the android library, gets compiled as an *'aar'* file. To use it in any application, the following steps must be followed:

1. It has to be copied to the *'project/app/libs'* folder of the app. The *'libs'* folder contains all the third-party libraries used by the app.
2. In the *'build.gradle(Module:app)'* we have to declare a dependency as *'implementation (name:'secureDroid', ext:'aar')'*.
3. In the *'build.gradle(Project)'* we have to declare a repository as *'flatDir {dirs 'libs'}'*.

After following the above steps, all the classes and methods of our wrapper are accessible to the app developer.

To make the application secure, the developers should use our wrapper's classes and API methods in place of the original classes and API methods. The use of the wrapper is straightforward as only a few lines have to be changed in the original code. e.g., for audio recording, we make an object of *'MediaRecorder'* class, but in place of this, we make an object of *'SecMediaRecorder'* class of our wrapper rest all the code and methods remain same. So overall, only one line of code is changed, which is not at all an overhead for the developer. Figure 4.1 shows the steps to integrate the wrapper into an application.

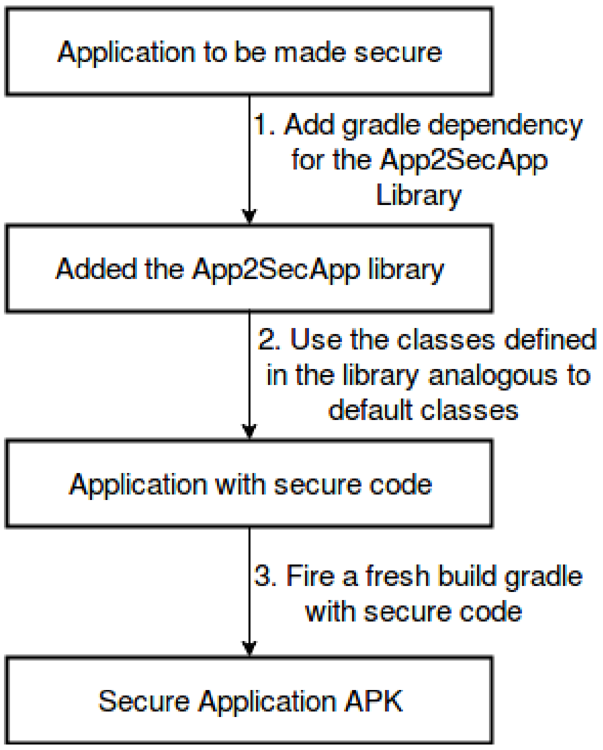

Application to be made secure
1. Add gradle dependency for the App2SecApp Library
Added the App2SecApp library
2. Use the classes defined in the library analogous to default classes
Application with secure code
3. Fire a fresh build gradle with secure code
Secure Application APK

Figure 4.1: Steps to Add Support for App2SecApp Wrapper

Chapter 5

Implementation Structure

This chapter explains the implementation structure and details of our wrapper.The wrapper is implemented as an android library. The implementation of our library is around 2300 lines of code. The library contains extended classes for all the classes which are used for the communication to and fro the applications. Such classes include MediaRecorder, Telephony Manager etc. Our library replaces the default classes with the corresponding extended classes provided by the library. The 'apk' generated after integrating the library in the app will make the app secure, giving the real-time monitoring of the app. The library contains mainly two modules:

- **Database Module**
- **Extended Classes Module**

5.1 Database Module

The database module is in charge of maintaining all the permissions which have influenced the app. We have used *'SQLite'*[20] Database for implementing this module. All the extended classes use this module to add and fetch the permissions residing in the DB. For this purpose, we implemented a 'DBHelper' class, which contains the following important methods:

- **insertPermission():** Used for inserting a permission into the DB.

- **getAllPermissions():** Used for fetching all the permissions residing in the DB which have influenced the app.
- **alradyExist():** Used for checking whether the permission we are trying to insert is already present in the DB. If it already exists, then nothing is done. Otherwise, it is added to the DB.

5.2 Extended Classes Module

This module is the main component; it contains all the necessary classes which have a role in communicating to and fro of apps sandbox. Following are the classes extended by the library:

- **CallLog, Camera, MediaRecorder, TelephonyManager, ContentResolver, Context, CursorLoader, Environment, LocationManager, SmsManager, Intent, URLConnection, SensorManager**

Each class name in our library has the same class name as of its original library prepended with 'Sec.' i.e. 'CallLog' class becomes 'SecCallLog' class in our library, similarly 'Camera' class becomes 'SecCamera' class in our library and so on. Each class includes all the sensitive API methods corresponding to its default class. The classes can contain the source or sink API method. A method may do some preprocessing depending on its type(Source or Sink API method). Figure 5.1 shows the block diagram of an app integrated with wrapper. We demonstrate an example of each method type with its corresponding class.

5.2.1 Source API Method

We implement *'SecLocationManager'* class corresponding to *'LocationManager'* class in our library for accessing data related to Location of the user. It contains the following sensitive methods:

- **isProviderEnabled()**

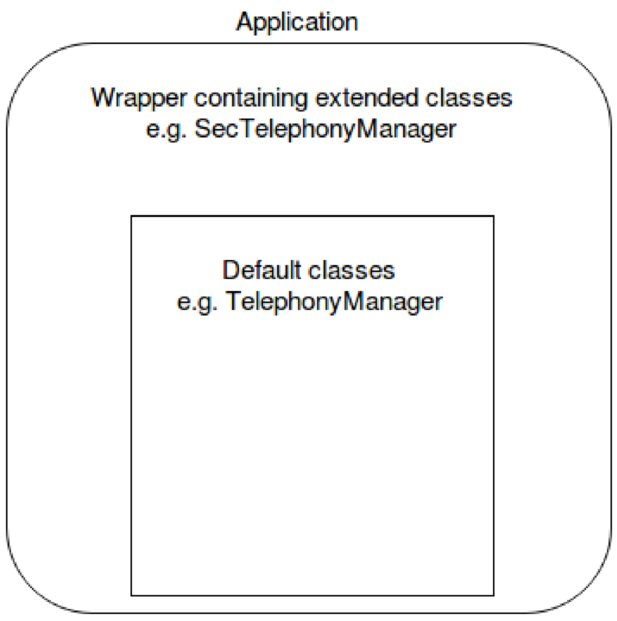


Figure 5.1: Block Diagram of an app integrated with wrapper

- **getLastKnownLocation()**
- **requestLocationUpdates()**
- **getProviders()**

The *'getLastKnownLocation()'* method is a source API method.It does preprocess before calling the actual API call. It inserts *'Location'* permission in DB through following code snippet:

```
DBHelper myDb = new DBHelper(context);
myDb.insertPermission("Location");
```

Other examples of source API methods include *'createCaptureSession()'* in *'SecCamera'* class which inserts *'Camera'* permission in DB, *'getLastOutgoingCall()'* in *'SecCallLog'* class which inserts *'Read_Call_log'* permission in DB, etc.

5.2.2 Sink API Method

We implement *'SecSmsManager'* class corresponding to *'SmsManager'* class in our library for sending SMS/MMS to a person. It contains the following sensitive methods:

- **sendTextMessage()**
- **sendMultimediaMessage()**
- **sendDataMessage()**

The *'sendTextMessage()'* method is a sink API method.It does preprocess before calling the actual API call. It retrieves all the permissions residing in the DB and prompts the user with a warning message through following code snippet:

```
myDb = new DBHelper(context);
allPermissions = myDb.getAllPermissions();
final Handler handler = new Handler(){
    public void handleMessage(Message mesg){
        throw new RuntimeException();
    }
};
AlertDialog.Builder alert = new AlertDialog.Builder(context);
alert.setTitle("Are you sure?");
alert.setMessage("The application has been influenced by following
      permissions recently:\n" + allPermissions +
      " It wants to send a message.");
alert.setPositiveButton("Continue", new DialogInterface.OnClickListener(){
    public void onClick(DialogInterface dialog, int id){
        sendSMS();
        Toast.makeText(context, "SMS sent", Toast.LENGTH_SHORT).show();
        handler.sendMessage(handler.obtainMessage());
    }
});
alert.setNegativeButton("No", new DialogInterface.OnClickListener(){
    public void onClick(DialogInterface dialog, int id){
        Toast.makeText(context, "SMS not sent", Toast.LENGTH_SHORT).show();
        handler.sendMessage(handler.obtainMessage());
    }
});
alert.show();
try{ Looper.loop(); }
catch(RuntimeException e){}
```

To stop the UI from running, we have used 'Handler.' Stopping the UI is necessary as the app should wait until the user gives a proper input, i.e., whether the user wants to continue this sensitive API call or wants to abort it.

Other examples of sink API methods include *'getOutputStream()'* in *'SecURLConnection'* class which prompts the user that "It wants to write some data to a server", *'getExternalStorageDirectory()'* in *'SecEnvironment'* class which prompts the user that "It wants to access the external storage", etc.

5.3 Efforts for the Implementation

For implementing our approach, we first made efforts for finding all the classes that are responsible for handling the permissions given to the applications. There could be multiple classes accountable for accessing the data related to particular permission, so extending each of them was necessary by our library, making all the communication channels monitored. Secondly, we found all the sensitive API methods in those classes which were responsible for transferring this sensitive data to and fro the app's sandbox. We implemented pre-processing code for each of such methods depending upon whether it is Source or Sink API method. The efforts done made the App monitoring easy giving us the required behavior of the Library.

Chapter 6

Case Studies

In this chapter, we have illustrated four case studies on which we evaluate our approach. We also point out the source and sink permissions and API calls used by the applications.

6.1 Case Study 1: GPS_SMS

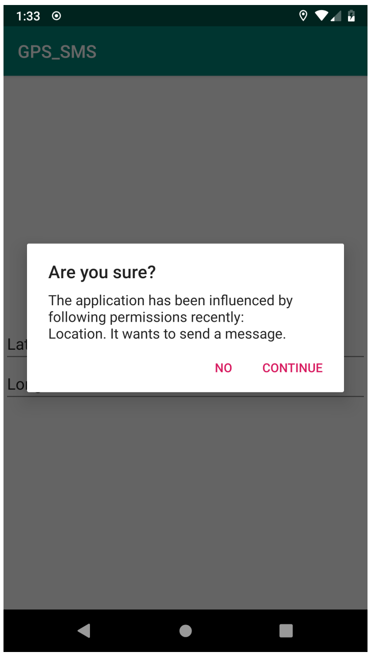
GPS_SMS is a malicious application which retrieves the location of the device and displays it to the user. It misuses the permissions given by the user and also sends a text message to an unknown number containing the details of the location of the user.

Even when the user gives the 'SMS' permission to the app unknowingly, our approach detects the malicious behavior of the app efficiently. The app works as:

1. Firstly, the source API method 'requestLocationUpdates()' is called for accessing the location of the User, so correspondingly 'Location' entry is made into the DB by our wrapper.

2. Now, whenever the app calls the sink API method 'sendTextMessage(),' our wrapper checks the DB and prompts the user that your application is influenced by Location permission and now is trying to send the SMS to an unknown number. It indicates the user that his/her data of the location is being sent outside of its sandbox.

The prompt message is shown in fig. 6.1a and if the user selects continue the SMS with location is sent as shown in fig. 6.1b. The table 6.1 and table 6.2 shows the source and sink permissions and API calls of the application respectively.

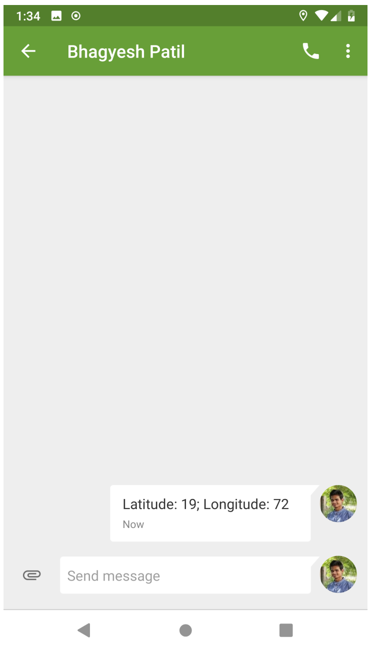

1:33

GPS_SMS

Are you sure?

The application has been influenced by following permissions recently: Location. It wants to send a message.

(a) Prompt message


1:34

Bhagyesh Patil

Latitude: 19; Longitude: 72

Now

(b) Sent SMS

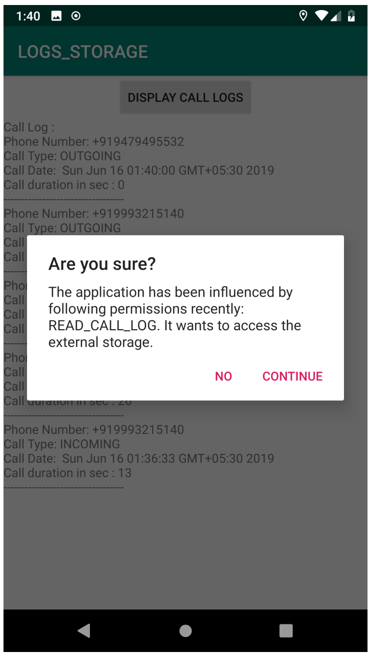
Figure 6.1: Snapshots taken from the GPS_SMS application

6.2 Case Study 2: LOGS_STORAGE

LOGS_STORAGE is a malicious application which accesses the call logs of the device and displays it to the user. It also tries to save the call logs to external storage, which can be then accessed by the outside world, thus exposing the user's private data.

Our approach works fine with this app and successfully detects it's malicious behavior, even when the user gives the 'Storage' permission to the app unknowingly. It prompts the user when the app tries to store the call logs into a file on external storage, thus indicating the user that his/her data of the call logs is being saved on external storage.

The prompt message is shown in fig. 6.2a and if the user selects continue the contacts are saved in a file called 'LOGS.txt' on external storage as shown in fig. 6.2b. The table 6.1 and table 6.2 shows the source and sink permissions and API calls of the application respectively.

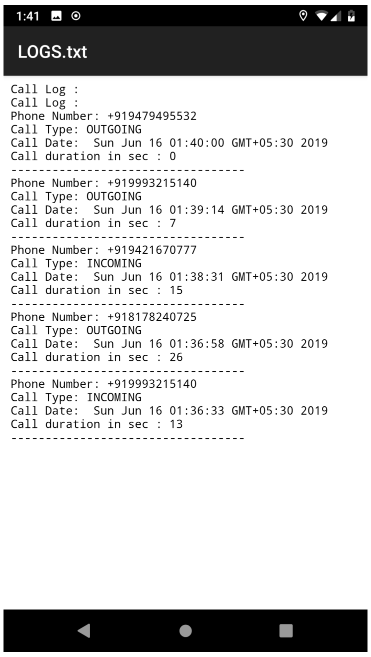


```
Call Log :
Phone Number: +919479495532
Call Type: OUTGOING
Call Date:  Sun Jun 16 01:40:00 GMT+05:30 2019
Call duration in sec : 0
--------------------------------
Phone Number: +919993215140
Call Type: OUTGOING
Call
Call
--------
Phon
Call
Call
Call
--------
Phon
Call
Call
Call duration in sec : 26
--------------------------------
Phone Number: +919993215140
Call Type: INCOMING
Call Date:  Sun Jun 16 01:36:33 GMT+05:30 2019
Call duration in sec : 13
--------------------------------
```

Are you sure?

The application has been influenced by following permissions recently: READ_CALL_LOG. It wants to access the external storage.

(a) Prompt message



```
Call Log :
Call Log :
Phone Number: +919479495532
Call Type: OUTGOING
Call Date:  Sun Jun 16 01:40:00 GMT+05:30 2019
Call duration in sec : 0
----------------------------------
Phone Number: +919993215140
Call Type: OUTGOING
Call Date:  Sun Jun 16 01:39:14 GMT+05:30 2019
Call duration in sec : 7
----------------------------------
Phone Number: +919421670777
Call Type: INCOMING
Call Date:  Sun Jun 16 01:38:31 GMT+05:30 2019
Call duration in sec : 15
----------------------------------
Phone Number: +918178240725
Call Type: OUTGOING
Call Date:  Sun Jun 16 01:36:58 GMT+05:30 2019
Call duration in sec : 26
----------------------------------
Phone Number: +919993215140
Call Type: INCOMING
Call Date:  Sun Jun 16 01:36:33 GMT+05:30 2019
Call duration in sec : 13
----------------------------------
```

(b) File stored in External Storage

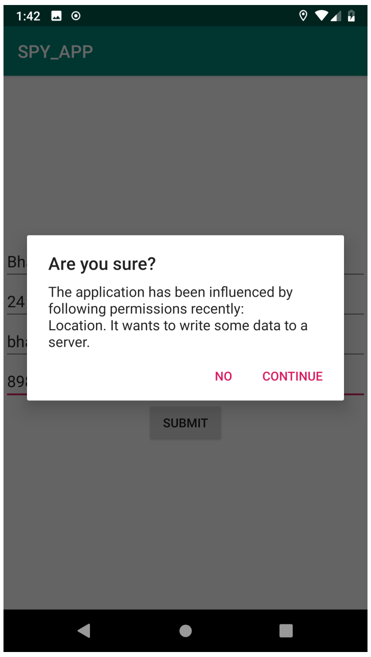
Figure 6.2: Snapshots taken from the LOGS_STORAGE application

6.3 Case Study 3: SPY_APP

SPY_APP is another malicious application created, whose proper functioning is to send the data of the form filled to the server, but instead of sending the form details it tries to send the location of the user to the server, thus exposing the user's private data.

We are able to detect this malicious behavior of the app through our wrapper. It prompts the user that the app has accessed the location of the user and is trying to send this data to a server.

The prompt message is shown in fig. 6.3a and if the user selects continue the Toast appears indicating connection established with the server as shown in fig. 6.3b. The table 6.1 and table 6.2 shows the source and sink permissions and API calls of the application respectively.

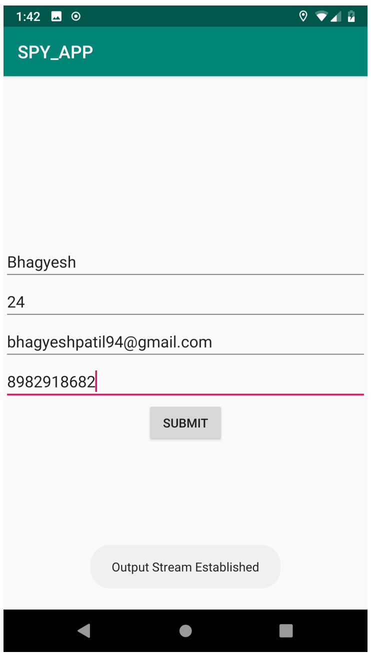


Are you sure?

The application has been influenced by following permissions recently: Location. It wants to write some data to a server.

NO CONTINUE

(a) Prompt message



Bhagyesh

24

bhagyeshpatil94@gmail.com

8982918682

Output Stream Established

(b) Toast indicating connection established

Figure 6.3: Snapshots taken from the SPY_APP application

6.4 Case Study 4: Multiple_Permissions

This app is created to demonstrate that our wrapper works well when dealing with multiple source and sink permissions. This malicious app has multiple activities. The first activity is a login page. After successful login, there are two activities.

- One activity asks the user to fill the form and to store that data on external storage, but instead of storing filled form information, the app maliciously stores calendars information on the external storage. The prompt message is shown in fig. 6.4a and if the user selects continue the calendars information is saved in a file called ‘Information.txt’ on external storage as shown in fig. 6.4b.
- Another activity displays the contacts of the user and tries to send an SMS to an unknown number having information of the last contact in the contact list of the user. The prompt message is shown in fig. 6.4c and if the user selects continue the SMS containing user’s last contact is sent as shown in fig. 6.4d.

Application	**Source Permission**	**Sink Permission**
GPS_SMS	LOCATION	SMS
LOGS_STORAGE	CALL_LOGS	STORAGE
SPY_APP	LOCATION	INTERNET
Multiple_Permissions	CALENDAR & CONTACT	STORAGE & SMS

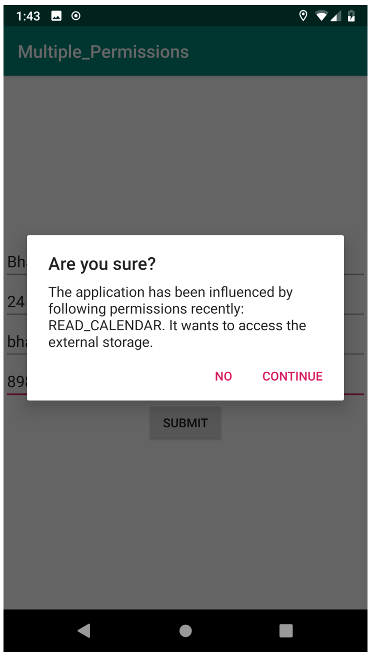
Table 6.1: Source and Sink Permissions of the malicious Apps

Application	**Source API Call**	**Sink API Call**
GPS_SMS	requestLocationUpdates()	sendTextMessage()
LOGS_STORAGE	secContentResolver.query()	secContext.getExternalFilesDir()
SPY_APP	requestLocationUpdates()	getOutputStream()

Multiple_Permissions	secContentResolver.query()
	secContentResolver.query()
		secContext.getExternalFilesDir()
		sendTextMessage()

Table 6.2: Source and Sink API calls of the malicious Apps

Even in the case of multiple source and sink API methods, our wrapper works fine and is able to detect the malicious behavior of the app. The table 6.1 and table 6.2 shows the source and sink permissions and API calls of the application respectively.

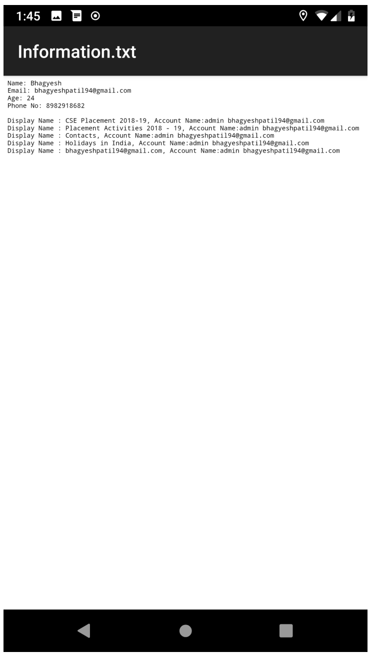

1:43

Multiple_Permissions

Are you sure?

The application has been influenced by following permissions recently: READ_CALENDAR. It wants to access the external storage.

(a) Prompt message of first activity

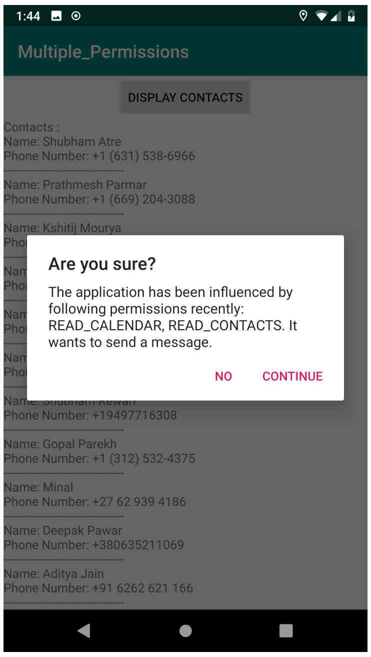

1:45

Information.txt

```
Name: Bhagyesh
Email: bhagyeshpatil94@gmail.com
Age: 24
Phone No: 8982918682

Display Name : CSE Placement 2018-19, Account Name:admin bhagyeshpatil94@gmail.com
Display Name : Placement Activities 2018 - 19, Account Name:admin bhagyeshpatil94@gmail.com
Display Name : Contacts, Account Name:admin bhagyeshpatil94@gmail.com
Display Name : Holidays in India, Account Name:admin bhagyeshpatil94@gmail.com
Display Name : bhagyeshpatil94@gmail.com, Account Name:admin bhagyeshpatil94@gmail.com
```

(b) File stored in External Storage


1:44

Multiple_Permissions

Contacts :
Name: Shubham Atre
Phone Number: +1 (631) 538-6966

Name: Prathmesh Parmar
Phone Number: +1 (669) 204-3088

Name: Kshitij Mourya

Phone Number: +19497716308

Name: Gopal Parekh
Phone Number: +1 (312) 532-4375

Name: Minal
Phone Number: +27 62 939 4186

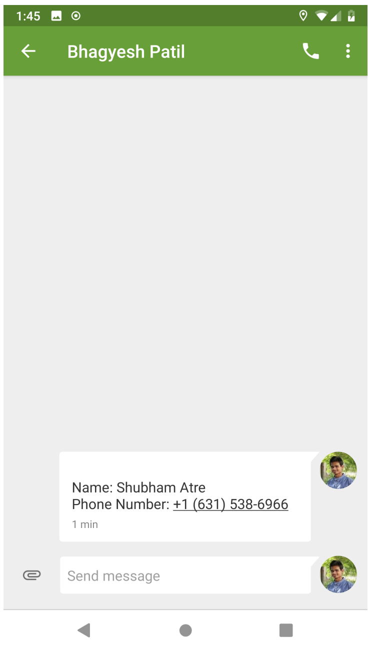
Name: Deepak Pawar
Phone Number: +380635211069

Name: Aditya Jain
Phone Number: +91 6262 621 166

Are you sure?

The application has been influenced by following permissions recently: READ_CALENDAR, READ_CONTACTS. It wants to send a message.

(c) Prompt message of second activity


1:45

Bhagyesh Patil

Name: Shubham Atre
Phone Number: +1 (631) 538-6966
1 min

(d) sent SMS

Figure 6.4: Snapshots taken from the Multiple_Permissions application

Chapter 7

Evaluation of Our Approach

Our evaluation of App2SecApp focuses on its ease of use, efficiency and user-friendliness. We define the ease of use as the efforts an application developer needs to put in, to avail the functionality of our wrapper. To quantify these efforts, we count the number of lines that needed to be changed for integrating the wrapper. We define efficiency as to whether the wrapper accurately identifies the source and sink API permissions, and correctly prompts the user. The user-friendliness is defined in terms of comparative feedback taken from a set of users interacting with the application before and after adding the App2SecApp library.

For evaluating App2SecApp, we select Pedometer[23], an open source Android application which uses hardware step detection sensor and storage permission. It counts the number of steps taken by the user and generates and stores the logs of it on external storage. The application is available on Google Play Store[22].

7.1 Ease of use

The first column of Table 7.1 depicts the LOC statistics for Pedometer. The total LOC is indicative of the project size. While calculating the LOC, we consider resource XML files, Java files for the application logic, and Gradle files for build. Intermediate compilation and settings files are omitted as these either not written by the developer or do not depend on the program log. The number of lines required to be changed depends on the number of classes used by the application that are responsible for communicating to and fro the

Language	**LOC Before**	**LOC After**	**Lines Modified**
Java	1963	1969	3
XML	716	716	0
gradle	108	110	0
Total	2787	2795	3

Table 7.1: LOC in Pedometer before and after adding the App2SecApp library along with number of lines modified

app.

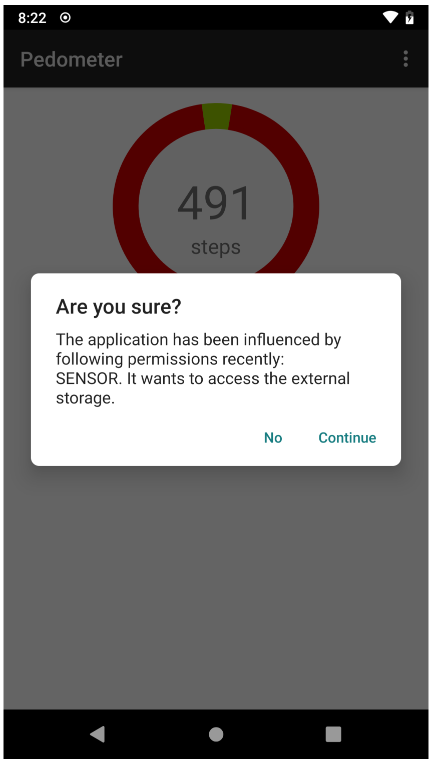
We count the LOC in Pedometer before and after adding the App2SecApp library along with the number of lines needed to be modified to add the library. These statistics show quantitatively how much effort is needed by the application developers to add support for App2SecApp.

7.2 Efficiency

Pedometer after integration of the wrapper behaves as expected. It correctly identify the Source and Sink API permission, and the user gets correctly prompted whenever the App tried to send the data outside its sandbox. Figure 7.1 shows the message prompt to the user while using modified Pedometer app.

7.3 User-friendliness

To check the user-friendliness of the library, we conducted a survey and asked the participants to use the original Pedometer application and the modified Pedometer application with the added functionality. A total of ten users were selected for the survey. Five of them had experience with programming whereas the other were from a non-technical background. The users were given a brief introduction as to how the functionality of the wrapper. After thoroughly using both the versions of Pedometer, the users were asked about the user-experience and user-interface. Users were satisfied in both categories. None of the users felt that the message prompts interferes too much with the host application; users were pleased as long as apps security was integrated.



Are you sure?

The application has been influenced by following permissions recently: SENSOR. It wants to access the external storage.

Figure 7.1: Prompt message in modified Pedometer app

Many applications have some implicit approvals of access for certain program execution path. The user's can be asked to provide the implicit paths for which the prompt will not be shown to the user. To improve the user-experience machine learning techniques can also be applied which will attempt to learn the inputs given by the user for a particular execution path or trace for a given number of times. And then after learning, the model can automatically hide the prompt by using the previously learned inputs for a specific user.

Chapter 8

Conclusion and Future Work

In this dissertation, we looked at the problem of the malicious behavior of the applications. The approach we follow is to provide real-time monitoring of the apps. The solution we build is generic enough to work across devices with different versions of androids. To the best of our knowledge, the currently available tools require third-party apps for their correct functioning, such as they require a third-party tool for dismantling the 'apk' file of the app. They also need access to the 'apk' files or code-base of the apps, and also they do not provide any real-monitoring of the apps.

We build a wrapper, an Android library that implements our approach of providing real-time monitoring of the app in a real environment when a user is using it. Our approach extends the classes containing sensitive API methods and keep tracks which information is coming inside and going outside the apps sandbox. We also created some simple and complex Android application to evaluate our wrapper and check it's functioning while the application is running. We consistently find our wrapper to perform satisfactorily every case. The implementation of our library is roughly around 2300 lines of code.

The existing applications such as [21], which are intended to provide security to the users, can also be made to use our wrapper for making them secure in terms of communication channels.

In our current implementation, we do not track exactly what data is being sent by the sensitive API calls. We also do not track exactly which API call leaks the data outside the app's sandbox. Also, the current implementation only detects intra-app malicious behavior as we want to provide real-time monitoring to the user. In the future, we will try to find the exact leakage point using our wrapper and will also try to find the precise data being leaked.

Bibliography

[1] "Android Framework". https://developer.android.com/guide/platform [Online; accessed 10-June-2019]

[2] "Android Permissions". https://developer.android.com/guide/topics/permissions/overview [Online; accessed 10-June-2019]

[3] "Google Maps". https://maps.google.com/ [Online; accessed 10-June-2019]

[4] "Facebook". https://maps.facebook.com/ [Online; accessed 10-June-2019]

[5] "Static Analysis". https://en.wikipedia.org/wiki/Static_program_analysis [Online; accessed 10-June-2019]

[6] "Dynamic Analysis". https://en.wikipedia.org/wiki/Dynamic_program_analysis [Online; accessed 10-June-2019]

[7] "Android Developer". https://developer.android.com/training/index.html [Online; accessed 11-June-2019]

[8] "Application Components". https://developer.android.com/guide/components/fundamentals [Online; accessed 11-June-2019]

[9] "Android Kernel Security". https://source.android.com/security/overview/kernel-security [Online; accessed 11-June-2019]

[10] "Android App Security". https://maps.facebook.com/ [Online; accessed 11-June-2019]

[11] Wu, Wen-Chieh and Hung, Shih-Hao. 2014. “DroidDolphin: a dynamic Android malware detection framework using big data and machine learning.” In *Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems.* Pages 247-252.

[12] Chang, S. 2013. “APE: A smart automatic testing environment for Android malware”. Dept. Comput. Sci. Inf. Eng., Nat. Taiwan Univ., Taipei, Taiwan, Tech. Rep

[13] Reddy, KP and Pareek, Himanshu and Patil, Mahesh U and others. 2015. “Dynaldroid: A system for automated dynamic analysis of Android applications.” In *Proceedings of the 2015 National Conference on Recent Advances in Electronics & Computer Engineering (RAECE).* Pages 124-129.

[14] Zheng, Cong and Zhu, Shixiong and Dai, Shuaifu and Gu, Guofei and Gong, Xiaorui and Han, Xinhui and Zou, Wei. 2012. “Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications”. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices.* Pages 93-104.

[15] Alzaylaee, Mohammed K and Yerima, Suleiman Y and Sezer, Sakir. 2016. “DynaLog: An automated dynamic analysis framework for characterizing android applications”. In *Proceedings of the 2016 International Conference On Cyber Security And Protection Of Digital Services (Cyber Security).* Pages 1-8.

[16] Asif Ali and R.K. Shyamasundar. 2017. “SecFlowDroid: A Tool for Privacy Analysis of Android Apps using RWFM Model”. MTech. Thesis.

[17] Prashant Maurya and R.K. Shyamasundar. 2018. “Security Analysis of Android Apps”. MTech. Thesis.

[18] Tiwari, Abhishek and Groß, Sascha and Hammer, Christian. 2018. “IIFA: Modular Inter-app Intent Information Flow Analysis of Android Applications”. In *arXiv preprint arXiv:1812.05380.*

[19] “Implicit Intents”. `https://developer.android.com/guide/components/intents-filters` [Online; accessed 16-June-2019]

[20] "SQLite Database". `https://www.sqlite.org/` [Online; accessed 14-June-2019]

[21] Patil, Bhagyesh and Vyas, Parjanya and Shyamasundar, RK. 2018. "SecSmartLock: An Architecture and Protocol for Designing Secure Smart Locks". In *Proceedings of the International Conference on Information Systems Security.* Pages 24-43.

[22] "Google Play Store". `https://play.google.com/store` [Online; accessed 18-June-2019]

[23] "Pedometer". `https://github.com/j4velin/Pedometer` [Online; accessed 22-June-2019]

Acknowledgements

I want to thank my guide and advisor, **Prof. R.K. Shyamasundar**, for his valuable guidance and regular discussions and advice throughout the course for the work. His insightful suggestions to the various problems that I faced during my project were not only useful but also helped me in broadening my basic understanding of the project area. I want to thank all the researchers whose works have been studied and referenced, which helped me to understand the problem better. I would also like to thank my colleagues Parjanya Vyas, Snehal Borse, Prateek Patidar, and Satyaki Sen, for their continuous support throughout my curriculum.

Signature: ....

Bhagyesh Patil

16305R003

Date: ...... June 2019