## USING ASPECTJ TO INFORM DEVICE OWNERS WHEN SENSITIVE PERSONAL DATA LEAVES APP

**A CULMINATING EXPERIENCE 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 IN COMPUTER SCIENCE**

**MATTHEW BERKMAN JULY 2020**

**Abstract**

Android’s permission system can prevent apps from directly accessing protected data they do not have permission to access, but it does not prevent apps from sharing the protected data they were granted access to. An app could unintentionally expose the user’s sensitive data to malicious sources. Android only requires that an app informs the user what permissions it needs and not how it utilizes the permissions. A tool is needed that can both inform users of sensitive data leak occurrences in an app and prevent data leaks from occurring. This paper presents a bytecode weaving approach that places security checks around potential data leakage points in an app at compile time. The values of the method call are examined for sensitive data before the method can execute. This project currently serves as an app developer tool and takes the approach of having the app developer and user specify what is sensitive data instead of going by Androids definition. This project was evaluated on two real-world apps to demonstrate its effectiveness in capturing sensitive data leaks.

**Keywords:** Bytecode Weaving; Android; Hybrid Analysis; AspectJ; App Developer Tool; Sensitive Data Leakage;

# Acknowledgements

I would like to express my gratitude to Professor Abeer AlJarrah, my project advisor and committee chair. This project would not have been possible without her input, support, and infinite patience.

I also want to express thanks to my second committee member Professor Hao Yue for showing interest in my project.

# TABLE OF CONTENTS

1. Introduction 6
2. Background 8
3. Aspect-Oriented Programming 8
4. Android’s Permission Model 8
5. Intents 9
6. Leaking Apps 9
7. Related Work 10
8. Motivation 11
9. Specification and Design 14
10. Implementation 15
11. Tools and Libraries Used 15
12. Sensitive Data 15
13. The Aspects 17
14. Leakage Points Handling 18
15. Leakage Points Not Yet Handling 19
16. Analyzation of Join Point Arguments 20
17. Searching for Sensitive Data in Join Point 24
18. Event Listener Mechanism 26
19. Case Study and Discussion 29
20. Case Study Tests 29
21. Limitations 30
22. Conclusion 32
23. Future Work 32
24. References 34
25. Appendix 37

Appendix 1 37

Appendix 2 38

# LIST OF LISTINGS

1. Listing 1. Getting Context from Activity 15
2. Listing 2. Sensitive Data File Example 16
3. Listing 3. Pointcut Example 18
4. Listing 4. Main Advice 18
5. Listing 5. Try with Resources Example 19
6. Listing 6. Snippet of Object Analyzation Process 22
7. Listing 7. Analyze All Parents 24
8. Listing 8. Searching for Sensitive Data 25

# LIST OF FIGURES

1. Figure 1: AspectJ Components 9
2. Figure 2: Permission Grouping 1 11
3. Figure 3: Permission Grouping 2 12
4. Figure 4: Permission Request 12
5. Figure 5: JSON Data Path Example 26
6. Figure 6: Example Sensitive Data Log File 28
7. Figure 7: Data Leak User Alert Choice 28
8. Figure 8: Data Leak Notification 28
9. Figure 9: False Positive Data Leak Event Example 30

# Introduction

Android is one of the most used mobile operating systems, with a market share of 74.13% as of December 2019 [1]. Android 1.0 was officially launched in 2008, with the Linux kernel serving as its foundation. Android has every application run in its own isolated process with its own resources and only allows communication between the applications through its secure inter-process communication (IPC) mechanism. If an application wants direct access to the device’s data or functionalities, it needs to declare, request, and be granted the permissions guarding those features. A permission will be granted either by the Android system or the user depending on the security level assigned to the permission. Since its initial launch, a new version of Android has been released at least once every year as of 2019. Each major iteration brought changes to the Android security model.

Despite the years of security updates, a user’s sensitive data is still not fully safe on Android. A malicious app could harvest the user’s data if the user grants it permission to access the data. However, a recent study [2] revealed malicious apps can circumvent Android’s permission system and access protected data through security holes or utilization of vulnerable privileged apps [3]. Android is lenient on data security enforcement for apps once they gain permission to the protected data. Whether a user’s sensitive data is secure is dependent on the security of the apps given access to the data.

For this project, a mechanism that catches and reports sensitive data leaks from a native Android app at runtime was developed. The mechanism performs compile-time bytecode weaving on the app to add security checks that prevent sensitive data leaks from going unnoticed. This project serves to inform users when an app is sending out sensitive data and help app developers patch up data leaks in their app. Additionally, this project allows app developers and users to specify what data should be considered sensitive so that any data can be secured; not just the data Android’s permissions protect. To test the effectiveness of this project in capturing sensitive data leaks, two real-world apps were modified to use this project. This project successfully exposed sensitive data being sent outside of the app’s environment for both apps.

The outline of this paper is as follows. Section 2 presents the needed background on AspectJ and Android. Section 3 reviews related work. Section 4 presents the motivation for the project. Section 5 provides an overview of the project and design. Section 6 is about the implementation of the project. Section 7 contains the results of the case study and discussion. Section 8 concludes this paper and presents future work.

# Background

## Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) has been created to capture design decisions that were not implemented by existing programming paradigms such as Object-Oriented & procedural programming [4]. These design decisions spread all over the code such as security checks, logging, and error-handling. These actions can be defined in *aspects* or *cross-cutting concerns* and then *weaved* into existing objects. AspectJ is the AOP extension for Java [5]. It enables defining additional implementation to run at certain well-defined points in the execution of the program (dynamic crosscutting mechanism). To be able to implement it, there are three main constructs:

* + - Join points are well-defined points in the execution of the program.
    - Pointcuts are a means of referring to collections of join points and certain values at those join points. It can be a predicate or expression that matches join points.
    - Advice are method-like constructs used to define additional behavior at join points. It is associated with a pointcut expression and runs at any join point matched by the pointcut. There are five advice types that define the time of execution: before, after, around, after returning, and after throwing.

Aspects are units of modular crosscutting implementation (similar to classes), composed of pointcuts, advice, and ordinary Java member declarations (Figure 1).

## Android’s Permission Model

Android prevents apps from accessing certain resources and device functionalities without permission. Android provides three types of permissions for apps: normal, dangerous, and signature [6]. Signature permissions are permissions an app creates for added security to its components. Normal permissions cover data and features that Android determined were a low risk to the user’s privacy or device functionality. The system automatically grants normal permissions an app request and prevents users from revoking them. Dangerous permissions cover data that Android deemed as sensitive user data (e.g. contacts, SMS, photos), and device features that could impact the user (e.g. camera, mic) [7]. Dangerous permissions are granted to the app by the user and can be taken away by the user.

A close up of a logo

Description automatically generated

## Figure 1: AspectJ Components

## Intents

Intents are what an app uses if it wants to request an action from its own components or an external component. Android provides two different types of intents: explicit and implicit [8]. An explicit intent is made when a specific component name is specified, thus declaring an exact destination for the intent message. Implicit intents are ones that do not and require the Android system to decide which component receives the intent. While explicit intents are typically meant for secure communication between an app’s own components, ones can be made to communicate with external components using the external app’s package name [9,10]. Both types of intents can be used to communicate with external and internal components.

## Leaking Apps

For this paper, there are only two types of apps, benign and malicious. Apps are considered malicious if they intentionally harvest sensitive data and send it to external sources without the user’s knowledge and/or permission. An app is benign if it does not send out sensitive data unless requested by the user, or unintentionally leaks the data to external sources.

# Related Work

In this section, previous work that addresses the problem of apps leaking what is deemed sensitive data to external sources is discussed. The works presented determine sensitive data leakage occurrences through either dynamic or hybrid analysis.

Tools such as TaintDroid [11] and AspectDroid [12] take the approach of only reporting the data leakage event to the user. TaintDroid is a well-known privacy leakage monitoring tool that has been expanded upon in other projects [13, 14]. AspectDroid is a hybrid analysis system that injects monitoring code into an app's bytecode using AspectJ. These projects dictate what data is sensitive and label the APIs used to access that data as taint sources and label the APIs used to send data outside of the app's environment as taint sinks. The projects also implement taint propagation monitoring between sources and sinks to ensure data manipulation does not circumvent their security checks.

Capper [15] takes it a step further and aims to prevent data leakage from occurring without the user’s approval. To not overwhelm the user, Capper stores the user’s decision for every data leak event in a second app. Capper will only prompt the user of data leaks if they are new, approved one-time-only, or denied before.

Systems such as the one proposed by Cha and Pak’s [16], attempt to prevent untrustworthy apps from accessing the sensitive data in the first place rather than preventing them from leaking the data they obtained. They use security policies designed by the user to determine whether an app receives real, blank, or fake contact data when the app queries for it from the Android system. Though the proposed security system would only work if the user categorized malicious apps as untrustworthy.

These dynamic analysis systems decide what data is considered sensitive and do not allow the app developer or the user to specify. Cha and Pak do propose a sensitivity level system for specific data, but they still decide the data the user can categorize. Data that does not fall under these system’s sensitive data guidelines remain insecure.

# Motivation

Android enforces no mechanism that allows its users to know how secure their data is once they grant any (benign or malicious) apps dangerous permissions to access and modify that data. A user’s sensitive data could be leaked without the user ever knowing.

Google has invested countless man-hours into the development of Android and its security since their purchase of it in 2005. One major feature of Android’s security framework is the permission system. As discussed earlier, users can grant, deny, and revoke dangerous permissions an app must request if the app wants direct access to the protected device functionalities or user’s data [17]. Despite this security feature, Android only suggests that the app should inform the user when it uses the granted permissions [18]. Additionally, Android condenses all its dangerous permissions into groups and has the users grant permission groups instead of individual permissions as shown in Figure 2 and Figure 3. This leads to not knowing the exact permissions an app is requesting when the permission request alert comes up as shown in Figure 4. This simplicity prevents users from revoking specific permissions such as read\_contacts or write\_contacts. An app can refuse to start if not granted its requested permissions, which leaves the user to either blindly trust that the app will not abuse its permissions or not use the app at all.

A screenshot of a cell phone

Description automatically generated

## Figure 2: Permission Grouping 1

A screenshot of a cell phone

Description automatically generated

## Figure 3: Permission Grouping 2

A screenshot of a cell phone

Description automatically generated

## Figure 4: Permission Request

Chebyshev reports 3,503,952 malicious installation packages were found for Android in 2019 [19]. Malicious apps with a sizable user base can be found on Android to this day; one such case being the popular TikTok app, which has been reported to be harvesting user’s data [20]. Malicious apps such as TikTok require dangerous permissions to be granted to them before they will function. However, there is still a way for malicious apps to harvest sensitive data without ever being granted such permissions by the user. Permission re-delegation/escalation attacks involve a malicious app utilizing a privileged app to gain access to the protected device functionalities and sensitive data [21]. This occurs due to the privileged app not implementing proper security measures around the permission protected APIs it uses and allowing data to be leaked. Reardon et al. [2] report finding “…evidence of covert- and side-channel usage in 252,864 versions of 88,113 different Android apps, all of them downloaded from the U.S. Google Play Store”. He and Li announce finding 324 apps from a dataset of 4031 real-world apps that were vulnerable to permission re-delegation attacks [3]. Android does provide app developers methods for securing an app’s IPC usage such as checkCallingPermission [22] but does not enforce their usage, unlike permission requests.

Simply put:

* Android allows its users to be left in the dark regarding how an app uses its granted permissions to access and modify the user’s data.
* Benign privileged apps could unintentionally leak sensitive data to malicious sources if the app developer does not implement proper security mechanisms to handle various data leakage attacks such as permission re-delegation.

Android has made it difficult for its users to know how secure their permission protected data is once they grant any app permission to access and modify that data. A mechanism is needed that can expose mishandling’s of granted dangerous permissions by an app to leak sensitive data to outside sources. Not all data a user may consider sensitive, such as text entered into an app, is protected by Android’s permission system. So, the mechanism should also expose the leaking of data deemed sensitive by the user. Additionally, the mechanism should be able to allow the average user to act upon these sensitive data leaks by alerting them when they occur.

# Specification and Design

This project is designed to prevent a native Android app from leaking what is deemed sensitive data to external sources without the user’s knowledge. The project allows app developers and users to specify what data should be considered sensitive. The current implementation of this project focuses on being a useful tool for app developers in helping identify and patch leakage points within their benign apps, along with informing users of sensitive data leaks.

The project accomplishes this by using AspectJ [23] and Archinamon’s plugin [24] to bytecode weave security checks into an Android app at compile-time. Archinamon’s plugin allows AspectJ to work on Android builds. Initially, the security checks were meant to only be applied around the calls to the methods used for communicating with external sources. Though experimentation revealed the need to weave other types of methods. The security checks involve searching for sensitive data within the arguments of the method call at runtime and constructing a data leak event if sensitive data was found. The project uses a custom listener system to handle data leak events. While the app developers can design and register their data leak event listeners, the project provides its own such as one that would prompt the user with an alert of the event. The project has the registered listeners decide whether to allow the data leak to occur or not.

# Implementation

## Tools and Libraries Used

* IntelliJ IDEA Ultimate 2019.3
* Gradle 5.6.1
* AspectJ 1.9.4
* com.android.tools.build:gradle:3.5.0
* com.archinamon:android-gradle-aspectj:3.4.0
* Java 1.8

## Sensitive Data

Since the project has the app developers and users specify what is sensitive data, the specified data needs to be saved. Currently, the project saves what is deemed sensitive data to a text file in the app’s internal storage. Android prevents other apps from directly accessing the files in an app’s internal storage without the app providing temporary access through file sharing [25]. While this approach is reasonably secure, future versions of the project will have the file for the user’s sensitive data preferences exist outside of the weaved app’s environment. This is so the users would only need to specify choices in one spot instead of for each app implementing this project.

## Listing 1. Getting Context from Activity

@Before("execution(\* android.app.Activity+.onCreate(..))")

public void setupContextReferenceActivity(JoinPoint joinPoint){

if (!gotContextReference) {

if(joinPoint.getThis() instanceof Context) {

DataStorage.createFile((Context)joinPoint.getThis());

//No need for the app developer to create file.

gotContextReference = true;

....

}else{

....

}

}

}

If an app wants to access its internal storage files, a context reference is needed from a living component of the app [26]. Listing 1 shows how the project obtains a context reference when an activity component is being created. Similar setups are done for the entry points of a broadcast receiver and service component. The reason for this is so the project will have access to the sensitive data file if the app is launched from one of those three components. This approach was taken over the project implementing a personnel component because it does not negatively interfere with the apps component life cycle system. The reason why the content provider component does not have this arrangement will be discussed in Section 6.3.2.

The contents of the sensitive data file are JSON. The formatting of the JSON is left up to the app developer, though there are some conditions so that the project can properly read the file.

* Must be a JSON-Object at the top level, not a JSON-Array.
* Byte arrays must be Base64 encoded to strings and stored in the file with the key name having the word ‘Encoded’ in it.
* Sensitive images need to have their byte array be Base64 encoded to a string and stored in the file with the key name having the word ‘Encoded’ in it.
* Must store the URI with a scheme, authority, and either part of or the whole path.

This project uses Android’s built-in JSON library so that only AspectJ and Archinamon’s plugin are needed to use this project.

Listing 2 shows an example of what the sensitive data file would look like after using the provided sample contacts data gathering method. The method only serves as an example and is not meant to be a helper method for retrieving all the user’s contacts data.

## Listing 2. Sensitive Data File Example

{

"Contacts": [

{

"Contact\_Photo": {

"PHOTO\_THUMBNAIL\_URI": "content:\/\/com.android.contacts\/contacts\/5\/photo",

"PHOTO\_THUMBNAIL\_EncodedImage": [long string]

},

"StructuredName": {

"Given\_Name": "Matthew",

"Middle\_Name": "Calvin",

"Family\_Name": "Berkman",

"Phonetic\_Given\_Name": "Math-hew",

"Phonetic\_Middle\_Name": "Cal-vin",

"Phonetic\_Family\_Name": "Berk-man"

},

"Nickname": {

"Name": "Matt"

},

"Organization": {

"Company": "GenericCompany",

"Title": "Employee"

},

"Phone": {

"Number": [

"(923) 456-7891",

"(222) 333-4444"

]

},

"SipAddress": {

"Sip\_Address": "AsipServer"

},

"Email": {

"Address": [

"mberkC2060@gmail.com",

"calvin10@gmail.com"

]

},

"StructuredPostal": {

"Street": [

"1234 Albert Lane",

"5678 Street Ave"

]

},

"Im": {

"Data": [

"matthewSkype",

"matthewAim"

]

},

"Website": {

"URL": "www.matthewCB.com"

},

"Note": {

"Note": "Is a master’s student at SFSU"

},

"Event": {},

"Relation": {}

}

]

}

The two main reasons why this project has the app developers and users decide what is sensitive data instead of going by Android's definition are as follows.

1. It allows the app developer to specify their app’s data as sensitive, so it does not get leaked without it being logged.
2. Users may have different thoughts on what data is sensitive.

## The Aspects

This project currently does not handle all the leakage points provided by Android; the ones it does handle are represented by pointcuts. Excluding the context reference pointcuts mentioned in the previous section, all the pointcuts are call pointcuts since not altering the Android system, only the app. These pointcuts are attached to the same advice (Listing. 4), since the same security check needs to be applied to all of them for the time being. Different advice will be created in the future to reduce false positive data leaks. The reason Around advice is used is because it allows the project to alter the value returned by the join point and control whether the join point executes or not. To ensure the project’s own files did not get weaved, pointcuts were created to exclude the project’s files from being weaved at compile time.

## Listing 3. Pointcut Example

@Pointcut("call(\* android.content.Context+.bindIsolatedService(..))")

public void contextBindIsolatedService() {

}

## Listing 4. Main Advice

@Around("(contextBindIsolatedService() || contexBindService() || contextSendBroadcast() || " +

//Other Pointcuts................................

"|| jsonWriterName() || jsonWriterSetIndent() || jsonWriterValue()) "+

"&& excludeOwnFilesFromWeaving() && excludeSDListeners()")

public Object mainAdvice(ProceedingJoinPoint joinPoint) throws Throwable {

return analyzerProcess.analyzationDecider(joinPoint);

}

## Leakage Points Handling

Android has numerous methods an app developer could use to send data to internal or external sources. Below is a general description of the methods the project currently handles.

* Methods for launching most components.
* Methods for writing to a stream, not flushing it.
* Parcel class methods involved with writing data to native.
* PendingIntent methods.
* Methods for inserting data into a content provider.
* Methods for passing data along the component chain.

The initial plan for this project was to just weave the methods responsible for sending data outside of the app’s environment, though experimentation revealed that would not be enough.

Parcel and PendingIntent are examples of cases where the methods for passing data into the object need to be weaved. This is due to those methods writing the data to native or a class the project cannot access. Analyzing these objects at an exit point method would not reveal sensitive data despite the sensitive data being passed to the object earlier.

For the data written to an OutputStream or Writer object to be sent to its destination, the flush and/or close method must be called. However as shown in Listing 5, there exist scenarios where the app developer does not need to call flush or close themselves for data to be sent off [27]. So weaving flush and close methods would not allow intercepting all the data leaks involved with writing to a stream. Additionally, the project does not erase the sensitive data found in the arguments of a weaved join point. The project only reports the data leak and allows or denies the method execution. Denying flush or close methods from executing would lead to sensitive data still being present in the buffer. That is why the project weaves the methods for storing data in OutputStream or Writer object instead of flush and close.

## Listing 5. Try with Resources Example

try(FileWriter = new FileWriter(fileName)){

fileWriter.write(sensitiveData);

}catch(IOException e){

e.printStackTrace();

}

## Leakage Points Not Yet Handling

The leakage points the project does not yet handle fall into at least one of the three categories below.

1. Methods that call the methods already being handled, so a pointcut for that method needs to be created once the method is found.
2. Methods in the Jetpack/AndroidX library system.
3. Methods that are still being researched to determine if they could still be used to send data outside of the app’s environment after Android 8.0

Part ways through the project’s development, we were forced to upgrade the work environment and start using the Jetpack library system instead of the support library. Future iterations of this project will incorporate the leakage points contained in this library system. As new versions of Android release, the project will need updates to incorporate the new methods and security changes.

Android’s attempts to fix security holes in its system is the reason for point three. A major example being the changes done to shared preferences in Android 7.0. Before Android 7.0, app developers could allow external apps access to private files such as shared preferences by marking the files with the flag MODE\_WORLD\_READABLE. This file flag was blocked from being used at all in Android 7.0+ [28]. Security updates in Android 7.0 also required the use of the content URI instead of the file URI to allow external components access to an app’s internal storage files directly and the shareable content URIs for the files are generated from the FileProvider class [29, 30, 31]. Though FileProvider does not provide an option for creating a content URI for the shared preferences files. So, there is no reason to weave shared preferences methods after Android 7.0. To avoid unnecessary data leak false positives, research into the leakage potential of a method is done before a pointcut is made for it.

As brought up in Section 6.2, the project currently is not handling the entry points for the content provider component. What is meant by this is the project currently is not handling the cases where external components query an app’s public custom content provider to get the data stored in a database. Content provider queries return a cursor object that requires invoking methods to retrieve the data returned by the query. Handling cursors along with analyzing the contents of a file will come in future versions of the project once a fast and reliable solution for handling cursors and files has been developed. For now, this project is handling the writes to these objects. The project assumes if the user is fine with the app storing sensitive data somewhere, they are fine with the app sending the data stored to other places.

## Analyzation of Join Point Arguments

When a method that one of the pointcuts captures is called, the advice is run instead and begins the join point analyzation process. The join point arguments/values that get analyzed for sensitive data are the method arguments and the object the method was called on. Each argument is analyzed for sensitive data one at a time so that an informative data leak event can be constructed. Before checking if an argument contains sensitive data, the primitives and strings need to be extracted from the argument.

The argument is passed to the project’s object deconstructor where three collections are initialized before the argument is examined.

* ObjectsLookedAt: An IdentityHashmap that is used to ensure an object/value is not examined more than once.
* ObjectsToAnalyze (the queue): Maintains a list of the values that need to be inspected before the analyzation process is complete. The queue grows and shrinks throughout the process.
* PrimAndStringValues: A HashMap that holds the primitive and string values retrieved from the argument. The values are stored in the map as the string keys, while the map-value is for holding the label assigned to the value.

An IdentityHashmap compares values by reference and does not use the objects equals and hashCode method. Do not want the object to have the ability to affect the analysis process.

Listing 6 shows the analyzation process structure. A first-in last-out approach was taken so that the values in the queue did not need to be shifted for every value removal. When a value is popped off the queue and determined to be unique, it gets deconstructed based on the type of value it is. In its current state, the project would place the value in one of six categories, primitives, string, array, URI, bitmap, or unhandled object. Future iterations of the project will handle other Android objects that require method calls to access the significant data, such as Cursors.

URI and bitmap have a category because the desired values need a specific approach to be properly obtained. For URIs, need to call toString on the object to obtain the complete URI since the individual URI elements are uninformative due to the format requirement of URIs in the sensitive data file. Android bitmaps represent an image and the data of the image, the pixel values, is stored on the native side. Need to utilize bitmap’s methods to retrieve the byte array of the image. Due to the complexity of images, the project only does basic handling of images, one to one matching. Byte arrays of images are required to be stored as Base64 encoded strings in the sensitive data file. The Base64 encoded string of the byte array is stored in primAndStringValues with the map-value ‘IsBitmap’ to indicate the value should be handled a special way during sensitive data checking. Due to the possibility of a string value being a string representation of a byte array, the string value is transformed into a byte array and put on the queue.

## Listing 6. Snippet of Object Analyzation Process

void analyzeObject(Object value) {

objectsToAnalyze.add(value);

Object aHolder;

while (!objectsToAnalyze.isEmpty()) {

aHolder = objectsToAnalyze.remove(objectsToAnalyze.size() - 1);

if (aHolder == null) {

Log.i("DeconstructorMap", "Got a null aHolder");

continue;

}

//Prevents entering infinite loop if object has reference to itself

if (objectsLookedAt.containsKey(aHolder)) {

continue;//already investigated object

} else {

objectsLookedAt.put(aHolder, new Object());

}

if (isWrapperType(aHolder.getClass())) {

if (!(aHolder instanceof Boolean)) {

//want to scrap boolean since wouldn't hold sensitive data

primAndStringValues.put(String.valueOf(aHolder), blank);

}

} else if (aHolder instanceof String) {

String aString = String.valueOf(aHolder);

byte[] bytes = aString.getBytes();

objectsToAnalyze.add(bytes);

} else if (aHolder.getClass().isArray()) {

analyzeArrays(aHolder);

} else if(aHolder instanceof Uri) {

Uri temp = (Uri) aHolder;

primAndStringValues.put(temp.toString(), blank);

}else if(aHolder instanceof Bitmap){

Bitmap temp = (Bitmap) aHolder;

ByteArrayOutputStream stream = new ByteArrayOutputStream();

temp.compress(Bitmap.CompressFormat.PNG,100,stream);

byte[] bytes = stream.toByteArray();

try {

stream.close();

} catch (IOException e) {

e.printStackTrace();

}

String encodedString=Base64.encodeToString(bytes,Base64.DEFAULT);

primAndStringValues.put(encodedString, "IsBitmap");

} else {//Is an object without a special handler.

//Deconstruct object through reflection

}

}

}

The array category has subcategories for the arrays, byte, char, other types/multidimensional. The char array is transformed into a string and put on the queue. The other types/multidimensional arrays are iterated through at the first dimension and each value is inserted into the queue after checking if it was not already looked at. Byte arrays produce two to three values that are inserted into primAndStringValues. This is because the byte array could represent three types of values, 1) a string, 2) array of bytes, 3) a Base64 encoded byte array.

* One is for two cases. Handling the strings that were transformed into bytes in the string category. Cases where sensitive data is sent out in its byte form, such as for OutputStream.write which only accepts bytes.
* Two is for handling the passing of sensitive byte arrays, such as images. The project requires sensitive byte arrays/images to be Base64 encoded to strings before being stored in the sensitive data file. So, the byte array is Base64 encoded and stored in primAndStringValues with the map-value ‘IsBase64Encoded’ to indicate the value should be handled a special way during sensitive data checking.
* Three is for handling cases where the sensitive data was Base64 encoded before being sent out. The byte array is Base64 decoded and inserted into primAndStringValues. Since not every byte array can be Base64 decoded, a try-catch block is used to catch invalid decoding attempts.

More encoding types would be implemented in the future.

The objects without a category are deconstructed through reflection. First, Fields of all the class/instance variables of the object and its parents are obtained by passing the object’s class reference to the method presented in Listing 7. Then while iterating through the list of Fields, the object is used to get the value of the field for that object’s state. This allows obtaining all the values in the object.

## Listing 7. Analyze All Parents

public List<Field> analyzeAllParents(Class<?> aValue) {

List<Field> objectsFields = new ArrayList<>(

Arrays.asList(aValue.getDeclaredFields()));

Class<?> parentClass = aValue.getSuperclass();

while (parentClass != null) {

objectsFields.addAll(new ArrayList<>(Arrays.asList(parentClass.getDeclaredFields())));

parentClass = parentClass.getSuperclass();

}

return objectsFields;

}

Once the queue is found to be empty, primAndStringValues is returned and passed to the data comparing class.

## Searching for Sensitive Data in Join Point

The project checks for sensitive data in a join point argument by comparing the strings from the argument data map and the sensitive data map. The data in the sensitive data file is retrieved and stored in a map before the first join point argument is deconstructed. If there is no sensitive data, there is no reason to analyze the join point arguments. The sensitive data map has the data as the map-key, and the map-value is a list that represents the JSON key-index path to that data (Figure 5). The list is used as data in the sensitive data leak event and used for checking if a piece of sensitive data was marked as ‘Encoded’.

Since sensitive data is searched by string comparison, all the keys of the sensitive data map have the spacing removed and all the characters are made lower case as shown in Listing 8. Each iteration through the keyset of the argument data map will produce a component of the sensitive data leak event, which will only get added if sensitive data was found within the key. The argument's data value is investigated for sensitive data in two different ways depending on the type of value it is.

The first way is if the argument data value has a specific string as its map value, then the argument data value is checked to see if it equals sensitive data that has the word ‘Encoded’ somewhere in its JSON key-index path. While this would handle other types of encoded data, it is primarily designed for handling leaking of sensitive images either by bitmap or byte array. Though the content URI of an image or the Base64 encoded string of an image can still be shared; those are handled in the second way sensitive data is checked.

The second way involves checking if the cleaned argument data value contains a cleaned sensitive data value. The reason the contains method is used is to handle cases where sensitive data has other values added on to it. An example of this is with a contacts full name. As shown in Listing 2, each part of a contact’s name is treated as a separate piece of sensitive data. URI string is handled in this approach because as mentioned in Section 6.2, the project allows storing URI with only parts of the path. This is so the app developer or user could specify that all files with URIs that start with a certain path should be considered sensitive data. The downside of the string contains approach is it could lead to possible false positives such as if the name ‘Matt’ were found in the Base64 encoded string of an image extracted from an argument. The project always sends out these types of data leak events to avoid false negatives. That is why the argument data value and the uncleaned sensitive data found are stored in the data leak event; the app developer and user will determine if it is a false positive or not.

## Listing 8. Searching for Sensitive Data

......

for (String sensitiveDataKey : sensitiveDataKeys) {

String cleanedData = sensitiveDataKey.replaceAll("\\s", "");

cleanedData = cleanedData.toLowerCase();

cleanedSensitiveDataMap.put(cleanedData, sensitiveDataKey);

}

Set<String> cleanSensitiveDataKeys = cleanedSensitiveDataMap.keySet();

for (String argDatakey : argDataMapKeys) {

//Clean argDataKey

OffendingData = new OffendingData(argDatakey);

for (String cleanedSensitiveData : cleanSensitiveDataKeys) {

if(argDataMap.get(argDatakey) instanceof String){

String temp = argDataMap.get(argDatakey).toString();

if(temp.equals("IsBitmap") || temp.equals("IsBase64Encoded")){

String theSensitiveData = cleanedSensitiveDataMap.get(cleanedSensitiveData);

List<String> thePath = sensitiveDataMap.get(theSensitiveData);

for (String s : thePath) {

if (s.contains("Encoded")) {

if (argDatakey.equals(theSensitiveData)) {

//Add argument data & sensitive data info to offendingData

}

break;

}

}

}

}

else if (cleanedKey.contains(cleanedSensitiveData)) {

//Add argument data & sensitive data info to offendingData

}

}

//Add offendingData value to offendingArgument if not empty

}

//Return offendingArgument if sensitive data was found.



## Figure 5: JSON Data Path Example

## Event Listener Mechanism

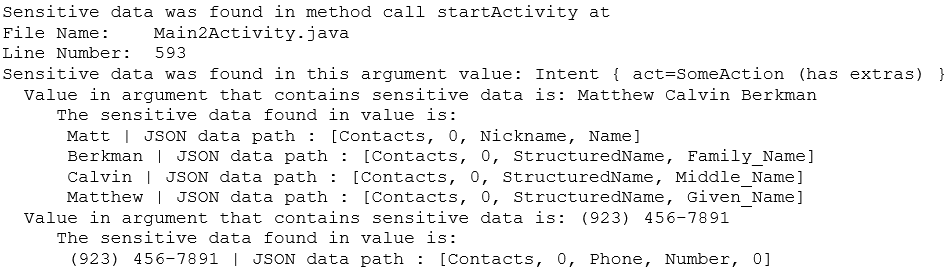
This project implements a custom listener registry for the sensitive data leak event. The sensitive data leak event contains info about the join point such as file line number, details of the arguments, and sensitive data found within the arguments as mentioned in the previous section. App developers can register custom listeners if they implement one of the two listener interfaces, both of which implement the same interface so that the registry remains simple. The two types of sensitive data leak listeners are

1. A listener that only needs the data leak event.
2. A listener that needs the data leak event and join point reference.

The listener registry passes the required data to each listener one at a time, waiting for a response before sending info to the next listener. The response is whether to let the join point execute or not. The registry works on a veto system; if one listener says no to letting the join point proceed, the project will not execute the join point. The veto system can be broken by the second type of listener though since it has the join point reference.

The project provides three sample listeners, one that logs the data leak event to a file (Figure 6), one that prompts the user with an alert choice (Figure 7), and one that sends out a notification to the user (Figure 8). The user alert listener is the main reason for the second type of listener. If the code being analyzed was executing off the UI thread, there is no issue. If the code was executing on the UI thread, then the user alert could break the app. Android does not display the user alert dialog until the UI thread returns to the main message queue, which requires the UI thread to continue executing the code after the join point currently being analyzed. So, the project must initially deny the join point from executing so the user can see the alert, and then on a separate thread, wait for the user’s response which will be the main decider on whether the join point executes or not. If the user approves of all presented data leaks, the listener utilizes the join point reference to execute the method. This works fine for methods such as startActivity, but methods such as writing to a stream will not have the intended outcome since the stream was most likely closed. A possible solution is to launch a second app to display the alert and pause the original app. The reason why this approach was not done for current implementation was because of Android’s main/UI thread policy. Android is strict on holding up the UI thread and will throw alerts at the user if the UI thread is held up for more than five seconds [32. 33]. Lastly, to initialize an alert, need an activity reference. The alert would only display on that activities view, even if it is not visible. The sample alert listener will not launch an alert if the attached activity is not visible, so unseen alerts do not accumulate.

The first type of listener is for the procedures that do not hold up the UI thread for long. The provided logging listener writes the data leak event info to a file in the app’s internal storage; an example is shown in Figure 6. This is meant to provide app developers a record of possible data leaks in their app, so they can fix them if they were not meant to happen. The provided notification listener just displays a sensitive data leak event in a single notification as shown in Figure 8.

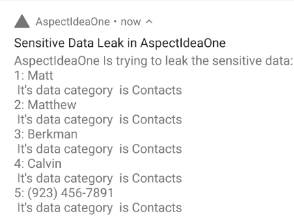


## Figure 6: Example Sensitive Data Log File

A screenshot of a cell phone

Description automatically generated

## Figure 7: Data Leak User Alert Choice



## Figure 8: Data Leak Notification

# Case Study and Discussion

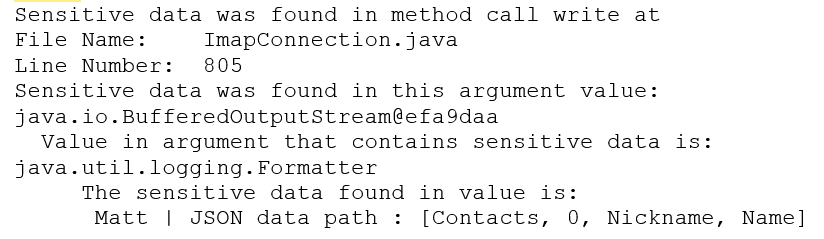
To test the effectiveness of this project on real-world apps, the project was attached to two real-world apps. The GitHub repositories of K-9 Mail [34] and Simple-Contacts Pro [35] were downloaded on July 14, 2020. K-9 Mail has over five million downloads on the Google Play store while Simple-Contacts Pro has over fifty thousand downloads on the Google Play store. To simulate the app developer specifying what is sensitive data, the example contacts gathering call was used. Additional tests should be conducted that specify a variety of data as sensitive, such as user input or apps own data. Both projects minSdk were increased to 26 and their Gradle version downgraded to 5.6 to be compatible with this project.

## Case Study Tests

The modified apps were manually interacted with on an emulated Android 8.0 device. The logging and notification sensitive data leak listeners were used for handling the reported sensitive data leaks. The listeners were set to allow any sensitive data leak to proceed after the leak was documented since wanted to test how the project affected the normal flow of the apps. For these tests, the contact data used is presented in Listing 2.

Simple-Contacts tests involve adding a new contact to our device using the app. The new contact only had the name ‘Calvin Jones’, with ‘Calvin’ being considered sensitive data since it is the middle name of the existing contact. The first test involved using this project without the prototype explicit intent checker, while the second test involved using the checker. The first test produced three alerts (Appendix 1) while the second test produced only two (Appendix 2). The extra alert in the first test was due to sensitive data being found when the app was launching an internal activity using an explicit intent. This outcome demonstrates the project has the capability of ignoring certain false sensitive data leaks at runtime. The second alert was for the app trying to write sensitive data to a content provider, which is considered an actual data leak by this project. The third alert, setResult, is a false positive that occurs due to the object method was called on containing sensitive data though that data is not used in the method execution. A solution for this is discussed in future work.

K-9 Mail test consists of logging in with an email marked as sensitive data and sending out an email that contains a phone number belonging to an existing contact. Several alerts were produced before and during email sending. Most of the alerts were from the app calling an OutputStream.write method with sensitive data in the arguments. They are all considered actual sensitive data leaks since the project considers any sensitive data writing to a stream as a sensitive data leak. Though some of them are logically false positives. Figure 9 shows one such event, where a small sensitive data value was found in an argument. These types of leaks can be dealt with through implementing a whitelist for Android’s values, which will be implemented as part of future work to reduce false positives.



## Figure 9: False Positive Data Leak Event Example

In terms of privacy leakage, the primary goal of this project was to intercept and expose sensitive data leaks within the app. While work is still being done on weaving exit points, the real-world app tests demonstrated the project is effective for the cases it currently handles. The secondary goal of this project is to limit false positives, so the app developers and users are not misinformed. The Simple-Contacts test showed this is possible.

## Limitations

* This project cannot weave dynamically loaded libraries or native code because Archinamon’s plugin does compile-time weaving and not load-time weaving, and only weaves java bytecode. This limits the data leakage points the project can analyze.
* Due to time constraints, the project was implemented onto apps found on GitHub instead of having the app developers utilize the project in their apps. This is a major limitation of the case study since the usability and usefulness of this app developer tool cannot be assessed.
* If sensitive data is altered, such as through encryption, before being sent out; the project would not be able to intercept it. A solution to this would be performing data propagation which is discussed in future work.

# Conclusion

Android does not enforce the need for app developers to inform a user how their app utilizes the permission protected data the app was granted access to by the user. An app could leak the user’s sensitive data without the user ever knowing. Malicious apps can use its granted permissions to harvest the user’s data. Even benign apps could unintentionally leak user’s sensitive data to external sources, such as a malicious app, if proper security measures are not employed.

In this paper, an app-level security mechanism was designed to expose sensitive data leaks from a native app during runtime. At its current stage, the project serves as an app developer tool to help highlight and prevent sensitive data leaks within their app. The project allows the app developers and users to specify what data should be considered sensitive, so even data not guarded by permissions can be secured. The project also allows app developers to design custom approaches for handling sensitive data leak events. The evaluation of this project on two real-world apps demonstrated the project’s ability to intercept data leaks in real apps.

## Future Work

Currently, work is being done to increase the amount of data leakage points the project handles. This mainly consists of creating a pointcut for methods that were identified as a way for data to be leaked to external sources. Certain leakage points, such as with content providers, require additional objects handlers in the deconstructor to properly analyze special Android objects such as cursors. On top of this, research is being done on ways to reduce false positives without causing false negatives. One direction being only analyzing the object method was called on if the object’s data is used in methods execution.

A necessary direction for this project is persisting user’s sensitive data preferences outside of the weaved app’s environment. This could improve the user’s experience since their sensitive data preferences would be in one spot instead of having one for each app. One way this could be achieved is by storing it in a custom app which could also provide a UI for the user to specify what data should be considered sensitive.

Another direction being investigated is the implementation of sensitive data propagation. This would prevent sensitive data from slipping through security checks due to being altered. Since app developers and users specify what is sensitive data, possible sources of the sensitive data need to be identified.

# References

1. S. O’Dea. 2020. *Market share of mobile operating systems worldwide 2012-2019*. Retrieved July 9, 2020 from https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/
2. Joel Reardon, Álvaro Feal, Primal Wijesekera, Amit Elazari Bar On,Narseo Vallina-Rodriguez, and Serge Egelman. 2019. 50 ways to leak your data: An exploration of apps’ circumvention of the android permissions system. In 28th {USENIX}*Security Symposium* ({USENIX}*Security* 19).603–620.
3. Y. He and Q. Li. 2016. Detecting and defending against inter-app permission leaks in android apps. In *2016 IEEE 35th International Performance Computing and Communications Conference* (*IPCCC*).1–7.
4. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin.1997. Aspect-oriented programming. In *ECOOP’97 — Object-Oriented Programming*, Mehmet Akşit and Satoshi Matsuoka (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 220–242.
5. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. 2001. An Overview of AspectJ. In *ECOOP 2001—Object-Oriented Programming*, Jørgen Lindskov Knudsen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 327–354.
6. Google LLC. *Protection Levels*. Retrieved July 9, 2020 from https://developer.android.com/guide/topics/permissions/overview#normal-dangerouss
7. Google LLC. *Determine sensitive data access needs.* Retrieved July 9, 2020 from https://developer.android.com/games/develop/permissions
8. Google LLC. *Intent types*. Retrieved July 9, 2020 from https://developer.android.com/guide/components/intents-filters#Types
9. Google LLC. *Receiving an implicit intent*. Retrieved July 9, 2020 from https://developer.android.com/guide/components/intents-filters#Receiving
10. Google LLC. *Building an intent*. Retrieved July 9, 2020 from https://developer.android.com/guide/components/intents-filters#Building
11. William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth.2014. TaintDroid: an information-flow tracking system for real time privacy monitoring on smartphones. *ACM Transactions on Computer Systems* (TOCS) 32,2 (2014), 1–29.
12. Aisha I Ali-Gombe, Brendan Saltaformaggio, Dongyan Xu, Golden G Richard III, et al. 2018. Toward a more dependable hybrid analysis of android malware using aspect-oriented programming. *computers & security* 73 (2018), 235–248.
13. Vaibhav Rastogi, Yan Chen, and William Enck. 2013. AppsPlayground: Automatic Security Analysis of Smartphone Applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy* (San Antonio, Texas, USA) (*CODASPY* ’13). Association for Computing Machinery, NewYork, NY, USA, 209–220. https://doi.org/10.1145/2435349.2435379
14. Lukas Weichselbaum, Matthias Neugschwandtner, Martina Lindorfer, Yanick Fratantonio, Victor Van Der Veen, and Christian Platzer. 2014. Andrubis: Android malware under the magnifying glass. *Vienna University of Technology, Tech. Rep*. *TR-ISECLAB-0414-001* (2014), 1–10.
15. MuZhangandHengYin.2014. Efficient, Context-Aware Privacy Leakage Confinement for Android Applications without Firmware Modding. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security* (Kyoto, Japan) (ASIA CCS ’14). Association for Computing Machinery, New York, NY, USA, 259–270. https://doi.org/10.1145/2590296.2590312
16. Youngrok Cha and Wooguil Pak. 2018. Protecting contacts against privacy leaks in smartphones. *PloS one* 13, 7 (2018), e0191502.
17. Google LLC. *Permissions overview*. Retrieved June 28, 2020 from https://developer.android.com/guide/topics/permissions/overview
18. Google LLC. [n.d.]. App permissions best practices. Retrieved June 28, 2020 from https://developer.android.com/training/permissions/usage-notes
19. Victor Chebyshev. 2020. *Mobile malware evolution 2019*. Retrieved July 1, 2020 from https://securelist.com/mobilemalware-evolution-2019/96280/
20. BBC News. 2019. TikTok sent US user data to China, lawsuit claims. *BBC News* (03 Dec. 2019). https://www.bbc.com/news/business-50640110
21. Adrienne Porter Felt, Helen J Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. 2011. Permission Re-Delegation: Attacks and Defenses. In USENIX *security symposium*, Vol. 30. 88.
22. Google LLC. *CheckCallingPermission*. Retrieved June 28, 2020 from https://developer.android.com/reference/android/content/Context#checkCallingPermission(java.lang.String)
23. Eclipse Foundation. Retrieved June 28, 2020 from https://www.eclipse.org/aspectj/
24. Ed Archinamon. 2020. android-gradle-aspectJ. https://github.com/Archinamon/android-gradle-aspectj
25. Google LLC. *Data and file storage overview*. Retrieved June 28, 2020 from https://developer.android.com/training/data-storage
26. Google LLC. *Context*. Retrieved June 28, 2020 from https://developer.android.com/reference/android/content/Context
27. Oracle. The *try-with-resources Statement*. Retrieved June 28, 2020 from https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html
28. Google LLC. *Save key-value data*. Retrieved June 28, 2020 from https://developer.android.com/training/data-storage/shared-preferences
29. Google LLC. *File system permission changes*. Retrieved June 28, 2020 from https://developer.android.com/about/versions/nougat/android-7.0-changes#permfilesys
30. Google LLC. *FileProvider*. Retrieved June 28, 2020 from https://developer.android.com/reference/androidx/core/content/FileProvider
31. Google LLC. *Sharing a file*. Retrieved June 28, 2020 from https://developer.android.com/training/secure-file-sharing/share-file
32. Google LLC. *Internals*. Retrieved July 9, 2020 from https://developer.android.com/topic/performance/threads#internals
33. Google LLC. *Threads*. Retrieved July 9, 2020 from <https://developer.android.com/guide/components/processes-and-threads#Threads>
34. Cketti. 2020. K-9. <https://github.com/k9mail/k-9>
35. Tibor Kaputa. 2020. Simple-Contacts. <https://github.com/SimpleMobileTools/Simple-Contacts>

# Appendix

## 1

Sensitive data was found in method call startActivity at

File Name: ContactsFragment.kt

Line Number: 13

Sensitive data was found in this argument value: com.simplemobiletools.contacts.pro.activities.MainActivity@4e1a8a

Value in argument that contains sensitive data is: Employee

The sensitive data found in value is:

Employee | JSON data path : [Contacts, 0, Organization, Title]

Value in argument that contains sensitive data is: GenericCompany

The sensitive data found in value is:

GenericCompany | JSON data path : [Contacts, 0, Organization, Company]

//Rest is just the other values in contact.//

Sensitive data was found in method call applyBatch at

File Name: ContactsHelper.kt

Line Number: 1309

Sensitive data was found in this argument value: [mType: 1, mUri: content://com.android.contacts/raw\_contacts, mSelection: null, mExpectedCount: null, mYieldAllowed: false, mValues: account\_type=com.google account\_name=mberkc2060@gmail.com, mValuesBackReferences: null, mSelectionArgsBackReferences: null, mType: 1, mUri: content://com.android.contacts/data, mSelection: null, mExpectedCount: null, mYieldAllowed: false, mValues: mimetype=vnd.android.cursor.item/name data6= data5= data4= data3= data2=Calvin Jones, mValuesBackReferences: raw\_contact\_id=0, mSelectionArgsBackReferences: null, mType: 1, mUri: content://com.android.contacts/data, mSelection: null, mExpectedCount: null, mYieldAllowed: false, mValues: data1= mimetype=vnd.android.cursor.item/nickname, mValuesBackReferences: raw\_contact\_id=0, mSelectionArgsBackReferences: null, mType: 1, mUri: content://com.android.contacts/data, mSelection: null, mExpectedCount: null, mYieldAllowed: false, mValues: data1= mimetype=vnd.android.cursor.item/note, mValuesBackReferences: raw\_contact\_id=0, mSelectionArgsBackReferences: null]

Value in argument that contains sensitive data is: Calvin Jones

The sensitive data found in value is:

Calvin | JSON data path : [Contacts, 0, StructuredName, Middle\_Name]

//All of the contact data found in the ContentResolver object method was called on.//

Sensitive data was found in method call setResult at

File Name: EditContactActivity.kt

Line Number: 1009

Sensitive data was found in this argument value: com.simplemobiletools.contacts.pro.activities.EditContactActivity@df7e110

Value in argument that contains sensitive data is: GenericCompany

The sensitive data found in value is:

GenericCompany | JSON data path : [Contacts, 0, Organization, Company]

Value in argument that contains sensitive data is: Calvin

The sensitive data found in value is:

Calvin | JSON data path : [Contacts, 0, StructuredName, Middle\_Name]

Value in argument that contains sensitive data is: content://com.android.contacts/contacts/5/photo

The sensitive data found in value is:

content://com.android.contacts/contacts/5/photo | JSON data path : [Contacts, 0, Contact\_Photo, PHOTO\_THUMBNAIL\_URI]

Value in argument that contains sensitive data is: Calvin Jones

The sensitive data found in value is:

Calvin | JSON data path : [Contacts, 0, StructuredName, Middle\_Name]]

// Rest of contact’s data being found in this object!//

Value in argument that contains sensitive data is: Calvin Jones

The sensitive data found in value is:

Calvin | JSON data path : [Contacts, 0, StructuredName, Middle\_Name]

## 2

Sensitive data was found in method call applyBatch at

File Name: ContactsHelper.kt

Line Number: 1309

Sensitive data was found in this argument value: [mType: 1, mUri: content://com.android.contacts/raw\_contacts, mSelection: null, mExpectedCount: null, mYieldAllowed: false, mValues: account\_type=com.google account\_name=mberkc2060@gmail.com, mValuesBackReferences: null, mSelectionArgsBackReferences: null, mType: 1, mUri: content://com.android.contacts/data, mSelection: null, mExpectedCount: null, mYieldAllowed: false, mValues: mimetype=vnd.android.cursor.item/name data6= data5= data4= data3= data2=Calvin Jones, mValuesBackReferences: raw\_contact\_id=0, mSelectionArgsBackReferences: null, mType: 1, mUri: content://com.android.contacts/data, mSelection: null, mExpectedCount: null, mYieldAllowed: false, mValues: data1= mimetype=vnd.android.cursor.item/nickname, mValuesBackReferences: raw\_contact\_id=0, mSelectionArgsBackReferences: null, mType: 1, mUri: content://com.android.contacts/data, mSelection: null, mExpectedCount: null, mYieldAllowed: false, mValues: data1= mimetype=vnd.android.cursor.item/note, mValuesBackReferences: raw\_contact\_id=0, mSelectionArgsBackReferences: null]

Value in argument that contains sensitive data is: Calvin Jones

The sensitive data found in value is:

Calvin | JSON data path : [Contacts, 0, StructuredName, Middle\_Name]

//All of contact data found in the ContentResolver object method was called on.//

Sensitive data was found in method call setResult at

File Name: EditContactActivity.kt

Line Number: 1009

Sensitive data was found in this argument value: com.simplemobiletools.contacts.pro.activities.EditContactActivity@def8248

Value in argument that contains sensitive data is: GenericCompany

The sensitive data found in value is:

GenericCompany | JSON data path : [Contacts, 0, Organization, Company]

Value in argument that contains sensitive data is: Calvin

The sensitive data found in value is:

Calvin | JSON data path : [Contacts, 0, StructuredName, Middle\_Name]

Value in argument that contains sensitive data is: Is a master's student at SFSU

The sensitive data found in value is:

Is a master's student at SFSU | JSON data path : [Contacts, 0, Note, Note]

// Rest of contact’s data being found in this object!//

Value in argument that contains sensitive data is: Calvin Jones

The sensitive data found in value is:

Calvin | JSON data path : [Contacts, 0, StructuredName, Middle\_Name]