Universidade de São Paulo
Instituto de Matemática e Estatística
Bachalerado em Ciência da Computação

Thomas Ryu Sameshima

A Study on How to Improve Android Permissions to the IoT Context

São Paulo
Novembro de 2017

A Study on How to Improve Android Permissions to the IoT Context

Monografia final da disciplina
MAC0499 – Trabalho de Formatura Supervisionado.

Supervisor: Prof. Alfredo Goldman

São Paulo
Novembro de 2017

Acknowledgement

First of all, I would like to thank my supervisor, Prof. Alfredo Goldman, whose knowledge, attention, and support made this work possible.

Secondly, I would like to thank Bruno Oliveira, a senior software engineer at Google, who clarified many questions about Android's permission system, even though it wasn't his area of expertise. His help was invaluable.

Thirdly, I would like to thank all my friends for keeping me company, specially Giovana Delfino, who helped me get through the smoothest internship approval process known to men, and Gustavo Zanon, who allowed me to use his impeccable LaTeX template.

And last but certainly not least, my family, for raising me in the most propitious environment, providing me everything that I could ever need and more.

The present work was carried out with the support of CNPq, Conselho Nacional de Desenvolvimento Científico e Tecnológico - Brazil

Abstract

Today, smartphones are sources of a great amount of personal information, e.g., the owner's contacts, photos, text messages, and current location. To prevent Android applications from freely obtaining such sensitive information, the system relies on permissions, a security mechanism which requires users to explicitly consent to an app's access to possibly harmful data. In this work, we describe the functioning of Android's permission system and its flaws, specially in an Internet of things (IoT) environment, and study ways to solve them.

Keywords: Android, Permissions, Privacy, Internet of things.

"Moon after moon did Berek make fools
Of the great and Untamed Three
Until malice for a Brother
Slew the hatred of the Other
And Berek did hunt
Alone and free."

Berek and the Untamed

Contents

Chapter 1

Introduction

1.1 Motivation

Privacy violation is one of, if not the most pertinent issue of modern civilization. From intense government surveillance (revealed, for example, by Edward Snowden's extensive leaks in 2013) to questionable corporate practices, personal information became not only a much sought after and valuable resource, but also a heavily threatened one.

Advancements in technology greatly helped us reach this dreadful situation. In his novel *Nineteen Eighty-Four (1984)* [1], George Orwell accurately predicted (back in 1949) we would have cameras in every building, watching our every move. What he couldn't predict, however, is that we would be constantly carrying one as well. Although considerable facilitators in our everyday lives, smartphones are currently one of the biggest threats to privacy. Their monitoring capabilities, combined with their ubiquity, make these devices sources of a huge amount of personal information. Information that many mobile applications today actively and aggressively try to obtain.

To make things worse, the popularization of Internet of things (IoT) caused a drastic increase in number of devices connected to the Internet and, more importantly, to smartphones (e.g., cars and home appliances). This increase, in turn, raised both the quantity of information they contain and the difficulty to protect it.

In such a complicated scenario, Android permissions are the operating system's only line of defense against exploitation. They attempt to protect the users' privacy by forcing applications to request for their authorization before using sensitive resources and data, such as the device's camera and stored files. Nevertheless, they ultimately fail in fulfilling their purpose, leaving users extremely exposed.

1.2 Objective

The objective of this work is to educate readers on how Android permissions work, determine the greater issues with the current permission system, and study, through changes made and proposed, ways to improve it.

1.3 Structure

The structure of this work will be as follows. Chapter 2 touches upon the subject of privacy, briefly defining it and describing the diversity present in the population's privacy concerns. Chapter 3 explains what permissions are, some of their characteristics, and how applications use them. Chapter 4 describe the issues with the current permission system based on chapter 2's definition, followed by some real-life examples. Chapter 5 presents possible alterations to Android's source code, which could solve the aforementioned issues. And chapter 6 discusses the population's lack of awareness, a problem much bigger than the permission system, but that ceaselessly plagues it.

Chapter 2

Privacy

2.1 Definition

In 1890, Samuel Warren and Louis Brandeis defined privacy as the “right to be let alone”, a necessity born from the advent of intrusive journalism, which sensitized the population to publicity and lower social standards and morality [2]. It was one of the first definitions ever written. Although vague, it helped give shape to an amalgam of incipient ideas, motivating the creation of numerous discussions and publications on the subject.

Not long after, in 1939, World War II started. It initiated an age of intense advancements not only in ballistics, but also in communication and espionage; humanity’s curiosity was further strengthened and stimulated by technology. In this context, the concept of privacy quickly shifted towards personal information, or it’s preservation and control, to be more precise. According to Alan Westin, privacy became “[...] the claim of individuals, groups, or institutions to determine for themselves when, how, and to what extent information about them is communicated to others,” [3] a definition made back in 1968 but still extensively used today.

2.2 Diversity

Studies show that one’s willingness to share personal information is heavily influenced by factors like cultural values, experience, and government involvement. They show, for example, that cultures with greater power discrepancies are less willing to share information, due to a higher mistrust for those in power; while cultures with greater collectivism are more willing, because they are more accepting of intrusions [4, 5]. In other words, these factors make society’s privacy concerns extremely heterogeneous, varying, in different degrees, from individual to individual. Therefore, when dealing with them, it is crucial to take this diversity into account.

Chapter 3

Android Permission

Android has a built-in security feature called the Android Application Sandbox, which isolates apps from each other and from the system itself, giving them a space with restricted access where they can operate safely. Permissions, as the name implies, are authorizations given to an app that allow it to access resources and data outside of it's own sandbox. They are Android's primary mechanism to address users' privacy concerns, by giving them control over which personal information is shared with an application.

3.1 Types of Permissions

There are two main types of permissions in Android:

- **Normal permissions**, which protect resources and data that offer very little to no risk to the user's privacy or to the execution of other apps. These permissions are automatically granted by the system when an application needs them.

  Some examples of normal permissions are:

  - `BLUETOOTH`: Allows the applications to connect to paired Bluetooth devices.
  - `BLUETOOTH_ADMIN`: Allows the applications to discover and pair Bluetooth devices on its own.
  - `INTERNET`: Grants the application access to the Internet.
  - `NFC`: Lets applications use near-field communication (NFC), a technology that allows the communication of two devices by bringing them close to each other.

- **Dangerous permissions**, which protect resources and data that may harm the user's privacy, affect the device's stored data, or interfere with the execution of other apps. They must be explicitly granted by the user to an application.

Some examples of dangerous permissions are:

- ACCESS_COARSE_LOCATION: Gives the application access to the approximate location of the device.
- ACCESS_FINE_LOCATION: Gives the application access to the precise location of the device.
- BODY_SENSORS: Grants access to data collected by health and fitness sensors.
- CAMERA: Grants access to the device's camera.
- RECORD_AUDIO: Grants access to the device's microphone.

3.2 Permission Groups

Dangerous permissions in Android all belong to permission groups. For example, both ACCESS_COARSE_LOCATION and ACCESS_FINE_LOCATION belong to the LOCATION permission group. When an application needs a dangerous permission, the system always requests its permission group to the user instead. If it is accepted, all further requests for permissions within that same group will be automatically accepted, without notifying the user .

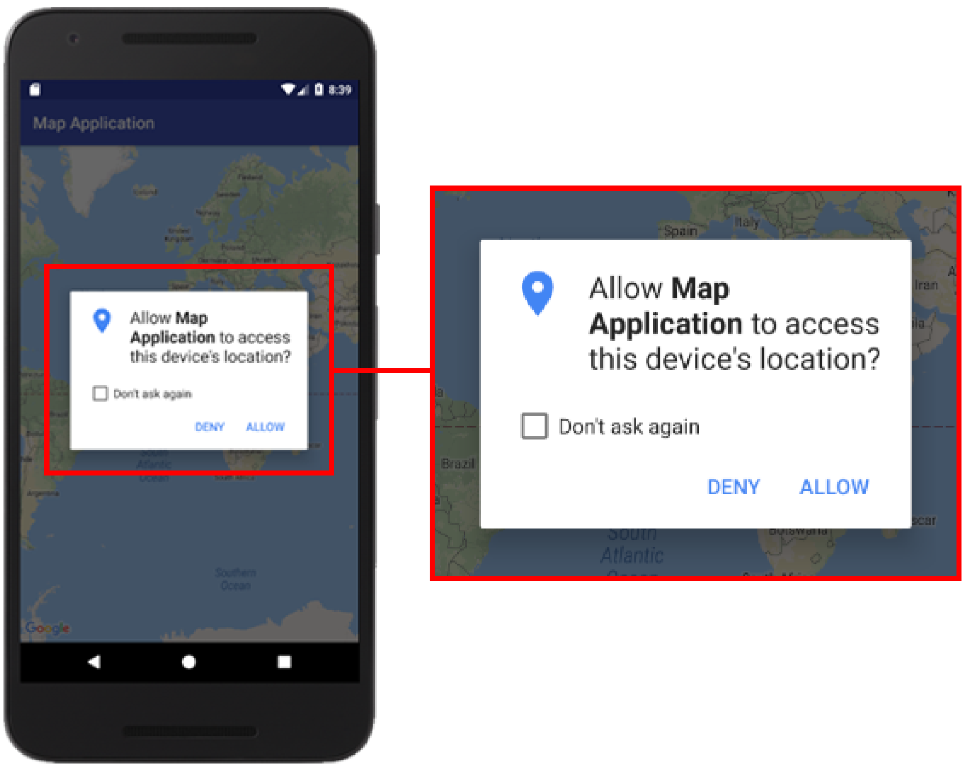


Figure 3.1: An application requesting the ACCESS_FINE_LOCATION permission (notice that the LOCATION permission group is requested to the user instead).

3.3 Changes with Android 6.0

Before the release of Android 6.0 Marshmellow, an application would request all the dangerous permissions it needed during the installation process. The user had to accept all of them if he wanted to install it on his device. Unfortunately, the only way of revoking these permissions, without using possibly harmful third-party apps, was by uninstalling it.

With its release, up-to-date applications now need to request each dangerous permissions separately during runtime, and the users can manage each of these permissions on the device's settings (allowing them to be enabled or disabled).

The probable motivation behind this new approach, besides speeding up the installation process, was to increase user security. Studies made before the release of 6.0 showed that an alarmingly low percentage of Android users payed attention to permissions when installing an app, and that an even lower number understood what each permission entitled the application to do; they concluded that Android permissions failed to inform the user of the privacy and security risks permissions imposed if used improperly [6, 7]. Consequently, these changes were most likely an attempt to remedy these problems.

3.4 Working with Permissions

There are four main steps an app developer must follow when using Android permissions:

1. Declare the permission.
2. Check if the application already has the permission.
3. If it doesn't, request the permissions to the user.
4. Handle the user's response.

3.4.1 Declaring the Permission

Before requesting any permission, it is necessary to declare all the permissions the application is going to use in its manifest, a file (`AndroidManifest.xml`) that contains essential information about the application.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.myappname">

    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.CAMERA" />

    <application ...>
        ...
    </application>

</manifest>
```

Code 1: An app declaring it's going to use the `INTERNET` and `CAMERA` permissions.

All normal permissions declared in the manifest will be automatically given to the app, without the user ever being notified about them.

If the device is running Android 5.1 or lower, *or* the application is configured to run in these versions, all of the declared dangerous permissions will be granted if the user agrees to install the application, and no further steps are necessary. Otherwise, if the device is running Android 6.0 or higher, *and* the application is configured to run in these versions, each dangerous permission must be requested separately to the user during its running (of course, taking permission groups into consideration).

3.4.2 Checking for Permissions

Since devices running Android 6.0 allow users to change an application's permissions in its settings, it is a good practice to always check if the app has a dangerous permission before using its related resources and data. To do this, a method called `ContextCompat.checkSelfPermission()` is used. It returns:

- `PackageManager.PERMISSION_GRANTED` if the application has the permission.
- `PackageManager.PERMISSION_DENIED` if the application doesn't have the permission, requiring it to be explicitly requested to the user.

```
if (ContextCompat.checkSelfPermission(this,
        Manifest.permission.CAMERA) == PackageManager.PERMISSION_GRANTED) {
        ...
}
else {
        ...
}
```

Code 2: A Java code that checks if the application has the `CAMERA` permission.

3.4.3 Requesting Permissions

To request one or multiple permissions to the user, the method `ActivityCompat.requestPermission()` is used, which takes as argument an array containing all the permissions to be requested and a request code, which is a integer that uniquely identifies the method call.

```
ActivityCompat.requestPermissions(this,
            new String[]{Manifest.permission.CAMERA}, 1234);
```

Code 3: A call which requests the `CAMERA` permission to the user, attaching the request to the code 1234.

With the method call, each dangerous permission that doesn't belong to a permission group already owned by the application will be requested via a prompt to the user (similar to the one displayed by figure 3.1) which, again, requests the permission group and not the permission itself.

3.4.4 Handling User Response

After the user replies to all requests made by an `ActivityCompat.requestPermission()` call, the system automatically calls the application's `onRequestPermissionsResult()` method, passing as arguments the request code of all the requests, the array with all the permissions requested, and an array containing the user's response for each permission listed in the previous array. The application must handle all responses (considering the possibility of multiple request codes) either by performing the operations related to the permissions (if their request was accepted) or by disabling them (if it was denied).

```java
@Override
public void onRequestPermissionsResult(int requestCode,
        String[] permissions, int[] grantResults) {

    /* If the user cancels the requests
       the results array stays empty. */
    if (grantResults.length > 0)

        switch(requestCode) {
            case 1234:
                if (grantResults[0] == PackageManager.PERMISSION_GRANTED) {
                    // Perform the operations related to the permission.
                }
                else {
                    // Disable the operations related to the permission.
                }
                break;
        }
}
```

Code 4: A implementation of the `onRequestPermissionsResult()` method, which the system will call after the user replies to a request sent by the `ActivityCompat.requestPermission()` call from code 3.

3.5 Permissions in Android Wear

Android Wear is Android's operating system for smartwatches and wearables, i.e., its IoT-related devices. Permissions in Android Wear work the same way as permissions in standard Android: they are either requested all at the same time upon installation, for devices running Android 5.1 or lower, or one at a time (when the application needs them), for Android 6.0 or higher. However, for Wear applications functioning as companions for phone applications, permissions granted to the latter are not shared with the former after Android 6.0. Simply put, a Wear app still needs to request dangerous permissions even if the accompanying phone app already has them.

Keep in mind an IoT device doesn't have to run Android Wear to communicate with an Android smartphone.

Chapter 4

Issues with the Current Permission System

While Android permissions are a powerful mechanism against invasive applications, who wish to prey on the users' personal information, it is not without its flaws. In this chapter, we will determine the issues with the current permission system using Alan Westin's definition of privacy, mentioned in section 2.1.

4.1 Issues

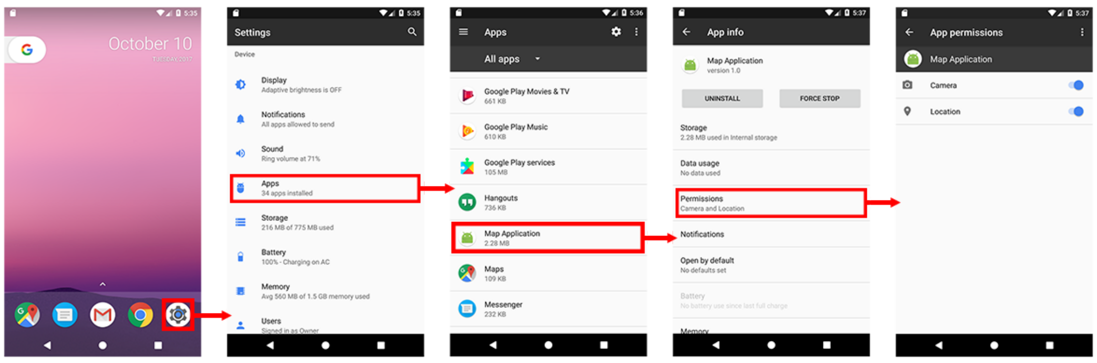
Based on Alan Westin's definition, three requirements must be met for privacy to be preserved. An individual must be able to "determine for himself", i.e., control:

1. *when* personal information is shared;
2. *how* personal information is shared;
3. *to what extent* personal information is shared.

4.1.1 When

If we were to reword this requirement into the permission system's context, it would be: a user needs to be able to dictate at what times an application has access to a permission's resources and data.

The only option currently given by the permission system is to enable or disable specific permissions in an application's settings, a feature brought with Android 6.0. It is a very impractical process that requires a user to navigate through multiple screens, definitely meant to be used sparingly (e.g., when he's not comfortable with a permission anymore). As a result, permissions are usually left enabled after their requests are accepted, allowing applications continuous access to possibly harmful information, even while they are running in the background or closed.


5:35
October 10
TUESDAY 2017

5:35
Settings
Device
Display
Adaptive brightness is OFF
Notifications
All apps allowed to send
Sound
Ring volume at 71%
Apps
34 apps installed
Storage
216 MB of 775 MB used
Battery
100% - Charging on AC
Memory
Avg 560 MB of 1.5 GB memory used
Users
Signed in as Owner

5:36
Apps
All apps
Google Play Movies & TV
661 KB
Google Play Music
610 KB
Google Play services
105 MB
Hangouts
736 KB
Map Application
2.28 MB
Maps
109 KB
Messenger
232 KB

5:37
App info
Map Application
version 1.0
UNINSTALL
FORCE STOP
Storage
2.28 MB used in Internal storage
Data usage
No data used
Permissions
Camera and Location
Notifications
Open by default
No defaults set
Battery
No battery use since last full charge
Memory

5:37
App permissions
Map Application
Camera
Location

Figure 4.1: The screens navigated to reach an application's permission settings.

4.1.2 How

To control how personal information is shared means to control "by what means" it is shared; namely, the tools used to collect and transmit it.

Android offers a plethora of permissions to give users ample control over some of the device's sensors, which are the primary tools to collect data (examples of these permissions are the aforementioned `CAMERA` and `RECORD_AUDIO` permissions). However, applications can still use a majority of these sensors unhindered, without having to declare or request any type of permission. The likely reason is these sensors, at first glance, seem way less intrusive than those protected by permissions. They include, for instance, light, humidity, and air pressure sensors. Nevertheless, Researchers from Newcastle University in the UK found that it is possible to use motion sensors, which are also permissionless, to inconspicuously discover a smartphone's PIN code by tracking the user's gestures [8]. It is not far-fetched to assume other types of sensitive information can also be obtained through them.

Additionally, the system automatically grants access to resources like Internet, Bluetooth, and NFC because they're tied to normal permission. They let applications freely send and receive data without the user's knowledge. This wouldn't be much of a problem if all sources of information were protected by permissions, because the user's explicit consent would still required in the end. However, we've seen that this is not always the case, being possible for applications to collect and transmit data unrestrictedly.

This issue can be specially hazardous in an IoT context, in which multiple electronic devices connect and communicate with each other and the Android smartphone. Non-Android/Android Wear IoT devices are not required to have anything akin to a permission system. Thus, an application can outsource the data collection to these device's sensors to free itself entirely from the constrictions of the permission system. For example, a GPS application can use its own brand of smartwatches to collect location data (protected in Android by dangerous permissions), Bluetooth to receive it from these devices, and Internet to send it somewhere else (e.g., a company's servers). Consequently, users can lose all their control over the process.

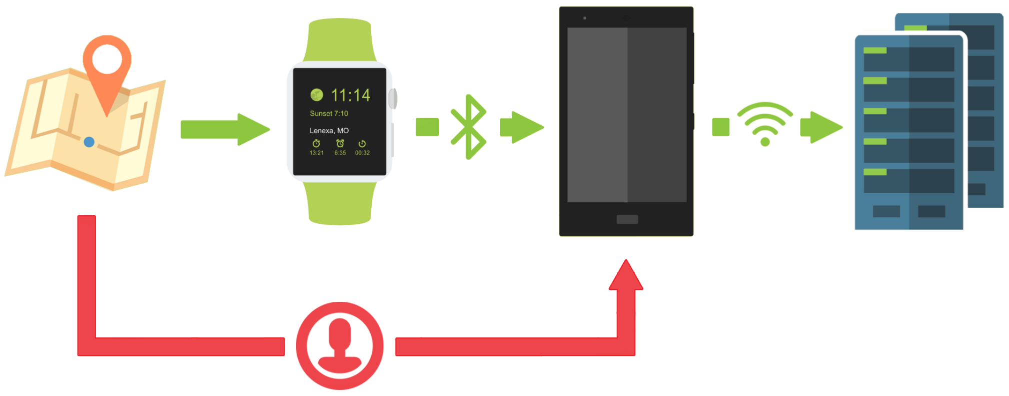


Figure 4.2: An application using non-Android/Android Wear IoT devices is not stopped by permissions.

4.1.3 To What Extent

The extent of information shared is understood not only by its quantity, but also its quality (i.e., its amount of detail). Ideally, the user should have flexible control over both of these parameters.

Currently, permissions either grants an application full access to its related resource or data, or completely denies them. They are a binary mechanism that work between two extremes, everything or nothing, neglecting the important diversity in the population's privacy concerns. Cautious users are forced to either deny/disable a permission, likely crippling the application's functionality, or entrust it with an uncomfortable amount, both in quantity and detail, of information (e.g., all of the files stored in the device or the user's exact location).

4.2 App Examples

4.2.1 Uber

Uber is the most well known ride-hailing application today, owning more than 80% of the market share in the US. Unfortunately, its renown is matched by its notoriety. In October 2016, for example, Uber's former forensic investigator Samuel Ward Spangenberg said in a court declaration that the company's "[...] lack of security regarding its data was resulting in [any] Uber employees being able to track high profile politicians, celebrities, and even personal acquaintances of Uber employees, including ex-boyfriends/girlfriends, and ex-spouses." [9] In other words, Uber exploits the openness of the permission system, which allows it to pervasively monitor users, whether they are using its services or not.

4.2.2 List Ease

List Ease is a shopping list application which possesses many of the features found in other applications of the genre, including the ability to:

- Add groceries to the list using an autofilling text input.
- Add quantities, prices and notes to each item on the list.
- Organize groceries into categories for an easier visualization.
- Share the list with other people, who can also add and tick off items from it.

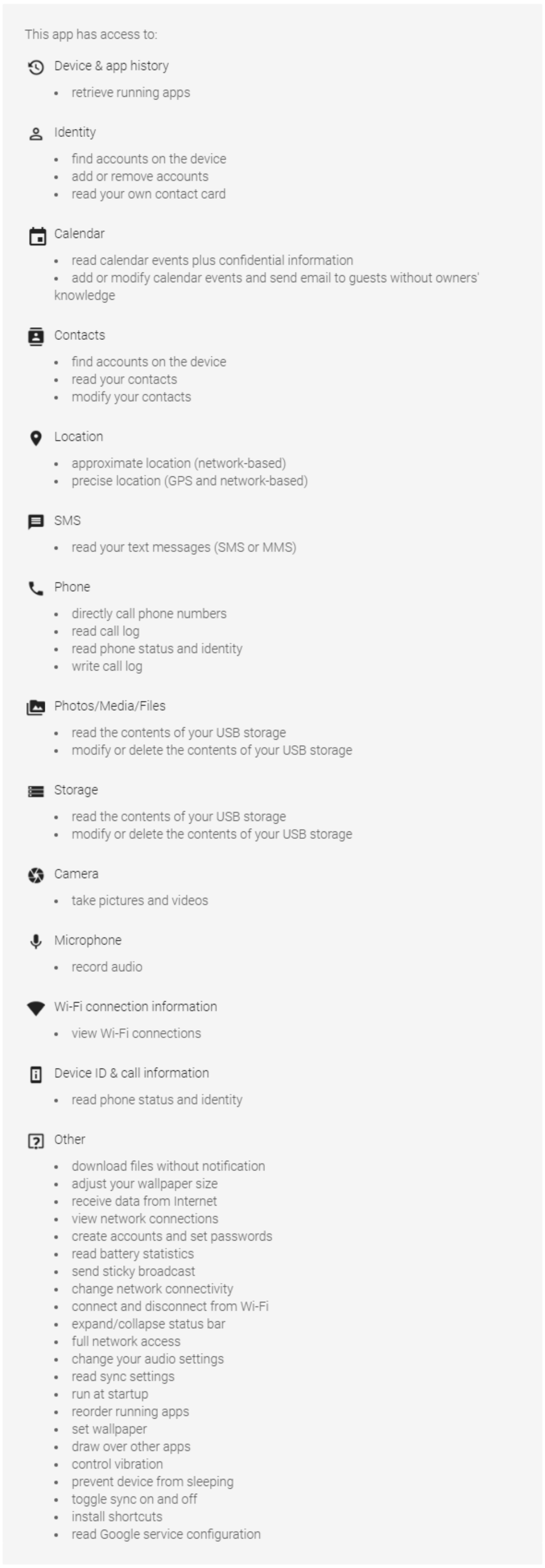
However, what makes this application so peculiar is that it uses Bluetooth LE (BLE) beacons to track with extreme precision the path taken by users inside stores that use this technology. With the excuse of sending users proximity-based reminders, inMarket, the company behind both List Ease and the beacons, combine Wi-Fi, GPS, and Bluetooth location data to form a rich database of what they call "foot traffic". Again, since Internet and Bluetooth are automatically given to the application, List Ease can inconspicuously connect to these beacons, receive the information they broadcast, and send it to inMarket. And the permission system does nothing in this situation.

Devices like the BLE beacons are a prime example of how difficult it is to protect personal information in an IoT environment with the current permission system. The application doesn't even need GPS (which is protected by permissions) to obtain accurate location data in this situation.

4.2.3 Facebook

Facebook's mobile app requests an absurd amount of permissions. It almost uses all of the dangerous permissions available in Android.

To address the users' concerns with this, Facebook has a dedicated page explaining the reason why some of these permissions are required. For example, the permission to read users' SMS messages is necessary because, according to them, it facilitates the process of linking their mobile numbers to their Facebook accounts. The problem here, nonetheless, is that Android still leaves applications with the granted permissions, even after their purpose has been served. As such, they can work like back doors, which were opened and forgotten by the user, and left ajar by the permission system, allowing applications to covertly obtain and modify information at any time. In Facebook's case, the amount of permissions given to it is so excessive it can basically hijack their users' devices at any time.



This app has access to:

Device & app history

- retrieve running apps

Identity

- find accounts on the device
- add or remove accounts
- read your own contact card

Calendar

- read calendar events plus confidential information
- add or modify calendar events and send email to guests without owners' knowledge

Contacts

- find accounts on the device
- read your contacts
- modify your contacts

Location

- approximate location (network-based)
- precise location (GPS and network-based)

SMS

- read your text messages (SMS or MMS)

Phone

- directly call phone numbers
- read call log
- read phone status and identity
- write call log

Photos/Media/Files

- read the contents of your USB storage
- modify or delete the contents of your USB storage

Storage

- read the contents of your USB storage
- modify or delete the contents of your USB storage

Camera

- take pictures and videos

Microphone

- record audio

Wi-Fi connection information

- view Wi-Fi connections

Device ID & call information

- read phone status and identity

Other

- download files without notification
- adjust your wallpaper size
- receive data from Internet
- view network connections
- create accounts and set passwords
- read battery statistics
- send sticky broadcast
- change network connectivity
- connect and disconnect from Wi-Fi
- expand/collapse status bar
- full network access
- change your audio settings
- read sync settings
- run at startup
- reorder running apps
- set wallpaper
- draw over other apps
- control vibration
- prevent device from sleeping
- toggle sync on and off
- install shortcuts
- read Google service configuration

Figure 4.3: All permissions used by the Facebook app (normal permissions are listed under "Other").

Chapter 5

Improving the Permissions System

In this chapter we will describe alterations that can and were made to Android's source code in an attempt to solve the issues presented in the previous chapter. All the files mentioned in this chapter are contained within Android's source files (which are compiled to generate its operating system), unless mentioned otherwise. For reference, build number N2G47Z was used to test all alterations. It runs Android version 7.1.2.

5.1 Changing a Permission's Type

Similarly to an application, Android also possesses a manifest. A file (`/framework/base/core/res/AndroidManifest.xml`) which, in this case, contains essential information about Android itself. All the available permissions are listed in it, followed by their attributes.

```
<permission android:name="android.permission.INTERNET"
    android:description="@string/permdesc_createNetworkSockets"
    android:label="@string/permlab_createNetworkSockets"
    android:protectionLevel="normal" />

<permission android:name="android.permission.CAMERA"
    android:permissionGroup="android.permission-group.CAMERA"
    android:label="@string/permlab_camera"
    android:description="@string/permdesc_camera"
    android:protectionLevel="dangerous" />
```

Code 5: The declarations of `INTERNET` and `CAMERA` in Android's own manifest.

As code 5 displays, `protectionLevel` is one of these attributes. It defines what type the permission is, and rewriting it is enough to change a normal permission into a dangerous one, and vice versa.

In section 4.1.2, we mentioned how an application's unrestricted access to communication technologies like Internet, Bluetooth, and NFC was a problem, because they allowed it to freely and covertly send and receive data. As a possible solution, we changed their associated permissions to dangerous permissions.

Such a simple alteration forces applications to request additional permissions before exchanging data with external devices. This is particularly helpful in preserving privacy in the IoT context, because it stops sensitive information from flowing so easily to and from the Android smartphone, diminishing the chances of it escaping the users' grasps.

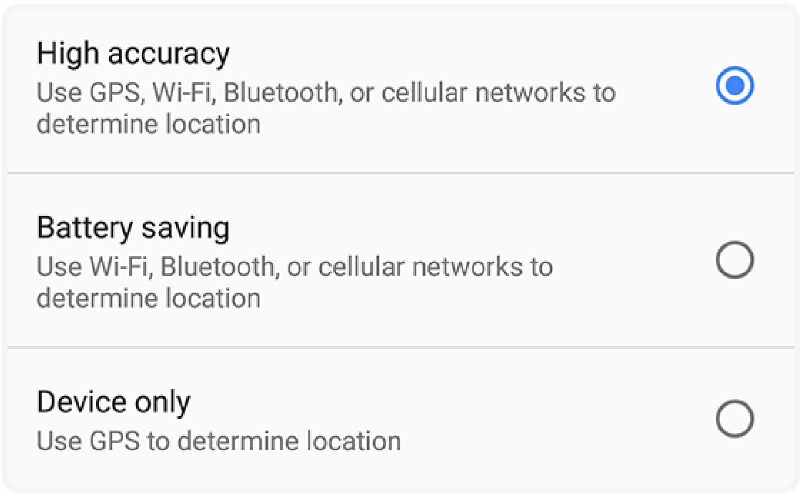
5.2 Setting Additional Restrictions

Another issue we have is the lack of flexible control over the shared information (i.e., when and to what extent it is shared). Users have no way to meaningfully limit an application's access to a permission's related resources and data, save for denying or disabling it. However, the system instead offers applications ample control over the received information in a majority of cases.

```java
final long MIN_TIME_PASSED = 10000;      // Time between location updates (in ms)
final float MIN_DISTANCE_TRAVELLED = 5;  // Minimum distance moved for location updates (in m)

private LocationManager mLocationManager;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    mLocationManager = (LocationManager) this.getSystemService(Context.LOCATION_SERVICE);

    // Checking and requesting ACCESS_FINE_LOCATION
    if (ActivityCompat.checkSelfPermission(this, Manifest.permission.ACCESS_FINE_LOCATION) ==
            PackageManager.PERMISSION_DENIED)
        ActivityCompat.requestPermissions(this,
            new String[]{Manifest.permission.ACCESS_FINE_LOCATION}, 1010);

    // Setting up LocationManager
    if (mLocationManager != null) {
        boolean isGPSEnabled =
                mLocationManager.isProviderEnabled(LocationManager.GPS_PROVIDER);
        if (isGPSEnabled)
            mLocationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER,
                    MIN_TIME_PASSED, MIN_DISTANCE_TRAVELLED, this);
    }
}
```

Code 6: An application setting up `LocationManager` to obtain the device's location. This process occurs when the appropriate screen is accessed.

`LocationManager`, for example, is an Android class that offers location services. Before using them, the application has to request one of the `LOCATION` group's permissions, and register to periodically receive location updates from a location provider. In code 6, for instance, the application chooses to request the `ACCESS_FINE_LOCATION`, use the GPS location provider (which possesses the highest accuracy), and registers itself to obtain the device's location whenever it moves 5 meters or 10 seconds passes. All these choices allow the application to determine how accurate the received information is and how frequent it is obtained, controls that users do not have. At most, they are allowed to broadly select which location providers are enabled to applications.



High accuracy
Use GPS, Wi-Fi, Bluetooth, or cellular networks to determine location
[✓]

Battery saving
Use Wi-Fi, Bluetooth, or cellular networks to determine location

Device only
Use GPS to determine location

Figure 5.1: The user's choices of which location providers to enable. Unfortunately, all of them provide location with relatively high accuracy.

We, then, gave users one of these choices. We allowed them to set the minimum time interval between location updates and the minimum distance travelled between location updates, determined by the `MIN_TIME_PASSED` and `MIN_DISTANCE_TRAVELLED` constants in code 6. They define how frequently location updates are received.

The place best suited to set restriction parameters like these is the device's settings. In most cases, the user's chosen values for them will override and, thus, ignore the application's. Nevertheless, for the frequency of location updates, both need to be taken into account.

Say the user determined the minimum time interval between updates is 30 seconds. If the application sets `MIN_TIME_PASSED` to a lower value, then the system should obviously override it with his choice; however, if the constant is set to a higher value, we would end up sending the app location data more frequently than necessary, hence it should not be overridden in this situation. Considerations like these are important to make when giving users new restriction options, since they can behave in various ways.

Therefore, the new steps followed by the `requestLocationUpdates()` method, which previously only registered applications for location updates, will be:

1. Receive the values set by the application for minimum time and distance.
2. Receive the values set by the user for minimum time and distance.
3. Compare them separately.
4. Register for location updates using the highest value of each.

To save the user's values we used the `Settings.System` class, which stores miscellaneous system preferences, such as text font size and screen brightness, in a simple name/value table. The application's values were still received as arguments by `requestLocationUpdates()`.

Our first step was to add both our new parameters (minimum time and distance) to the table by creating new constants and defining their default values. We chose zero for both, in order to make the application's choice always take precedence if the user's remain unchanged.

```
public static final class System extends NameValueTable {
    private static final float DEFAULT_FONT_SCALE = 1.0f;

    private static final float MIN_TIME = 0.0f;
    private static final float MIN_DIST = 0.0f;

    ...
}
```

Code 7: Adding our new parameters to the `Settings.System` class file (`/core/java/android/provider/Settings.java`).

Then, every time the user changed his values for the parameters in the device's settings, `Settings.System`'s `putFloat()` method was called to save them in the table.

```
float uMinTime = 0;
float uMinDist = 0;

... // Save the values set by the user inside the variables

final ContentResolver resolver = getContext().getContentResolver();
Settings.System.putFloat(resolver, Settings.System.MIN_TIME, uMinTime);
Settings.System.putFloat(resolver, Settings.System.MIN_DIST, uMinDist);
```

Code 8: Changing the minimum time and distance parameters. The `ContentResolver` object is used to, simply put, communicate with the name/value table.

To make `requestLocationUpdates()` receive both values from the table, we just had to call the class's `getFloat()` method. And to determine whether the user's or application's parameters should be used in the registration process, we compared them separately and settled for the ones with the highest value.

Note that `requestLocationUpdate()` is overloaded, which means it has multiple variants with different types of arguments. Here, we only changed code 6's version of the method, declared in `/location/java/android/location/LocationManager.java`; however all other versions should be altered as well.

```java
public void requestLocationUpdates(String provider, long minTime,
        float minDistance, LocationListener listener) {
    checkProvider(provider);
    checkListener(listener);

    LocationRequest request =
            LocationRequest.createFromDeprecatedProvider(provider,
            minTime, minDistance, false);
    requestLocationUpdates(request, listener, null, null);
}
```

Code 9: The original `requestLocationUpdates()` method.

```java
public void requestLocationUpdates(String provider, long minTime,
        float minDistance, LocationListener listener) {
    checkProvider(provider);
    checkListener(listener);

    final ContentResolver resolver = mContext.getContentResolver();
    long uMinTime = (long) Settings.System.getFloat(resolver, MIN_TIME, 0);
    float uMinDist = Settings.System.getFloat(resolver, MIN_DIST, 0);

    if (uMinTime < minTime) uMinTime = minTime;
    if (uMinDist < minDistance) uMinDist = minDistance;

    LocationRequest request =
            LocationRequest.createFromDeprecatedProvider(provider,
            uMinTime, uMinDist, false);
    requestLocationUpdates(request, listener, null, null);
}
```

Code 10: The modified `requestLocationUpdates()` method.

With this change, every time applications register themselves to receive location data through `LocationManager`, their preferences will first be compared with the user's. As a consequence, location updates will never occur more frequently than he's comfortable with. That is, personal information is provided more flexibly to applications.

Keep in mind we only added the minimum time and distance restrictions to one of the classes offering location services (a deprecated one to boot), the same way we only added it to one of the variants of `requestLocationUpdate`. To avoid applications bypassing a new restriction, it needs to be added all related classes and methods.

5.2.1 Other Restrictions

Other examples of restriction options that could be implemented are:

- **Location Data Accuracy:** In the beginning of this section, we've mentioned how applications are the ones who chose how accurate the received location data is. This is done by choosing to request either `ACCESS_COARSE_LOCATION` or `ACCESS_FINE_LOCATION`.

  When an application with `ACCESS_COARSE_LOCATION` attempts to obtain location data using `LocationManager`, it passes the information through `LocationFudger` beforehand. It is a class responsible for obfuscating the information, reducing its accuracy.

  It would be interesting to make users capable of determining whether `LocationFudger` also obfuscate the data sent to applications with `ACCESS_FINE_LOCATION`, and even how heavily its accuracy is reduced by changing the class's parameters through system settings.

- **When Location Access is Available:** iOS users can define when individual applications can track their location. They can choose between "Never", which is self explanatory; "Always", allowing tracking even while the app is closed or running in the background; or "While Using the App", requiring it to be running in the foreground.

  A restriction like this would greatly benefit Android users as well. It would stop the uncomfortable ubiquitous location tracking applications are allowed to have once permissions are granted, and increase the user's ability to establish when personal information is shared. This restriction could also apply to other types of resources and data (e.g., the devices camera, microphone, and stored data).

  To verify whether an application is running in the foreground, the `ActivityManager.getRunningAppProcesses()` method can be used. It returns a list of the app processes running in the device. Each process contains a `processName` and a `importance` attribute, which represents an application's name and the importance given by the system to it, respectively. A verification method could be as follows:

```
private boolean isAppInForeground (String appName) {
    ActivityManager manager =
            (ActivityManager) this.getSystemService(ACTIVITY_SERVICE);

    // Obtain the list of processes and iterate over it
    List<RunningAppProcessInfo> processes = manager.getRunningAppProcesses();
    for (RunningAppProcessInfo process : processes)
        if (appProcess.importance ==
                RunningAppProcessInfo.IMPORTANCE_FOREGROUND &&
                process.processName.equals(appName)) {
            return true;

    return false;
}
```

Code 11: A method which checks if an application is running in the foreground.

5.3 Creating New Permissions

5.3.1 Low-Level Permissions

Unix-like operating systems rely on user identifiers (UID) and group identifiers (GID) to determine which system resources are available to a user. Android, being a system built off of Linux, is no different. The Android Application Sandbox, for example, works by assigning a unique user identifier to applications and running each as their respective user in separate processes. It is similar to how, in a personal computer, all users have their own private set of files, which no other non-admin user can access.

Some permissions, hereafter called “low-level permissions”, assign specific GIDs to applications when granted, allowing access to their resources. The `BLUETOOTH` permission is one example of these permissions, let us see how it is implemented.

5.3.1.1 Manifest Declaration

As with any permission, BLUETOOTH is declared in AndroidManifest.xml. The same file mentioned in section 5.1, where all permissions are listed.

```
<permission android:name="android.permission.BLUETOOTH"
      android:description="@string/permdesc_bluetooth"
      android:label="@string/permlab_bluetooth"
      android:protectionLevel="normal" />
```

Code 12: The declaration of BLUETOOTH.

Besides the protectionLevel attribute, which we already know defines the type of the permission, the declaration must also have:

- label: briefly describes what the permission protects.
- description: a more detailed description than label.
- permissionGroup: an optional attribute which indicates the permission's group.

The first two attributes reference string resources, primarily located in the /frameworks/base/core/res/res/values/strings.xml file. There are many other different folders which store strings for other languages as well.

```
<string name="permlab_bluetooth">pair with Bluetooth devices</string>

<string name="permdesc_bluetooth" product="default">Allows the app to view
  the configuration of the Bluetooth on the phone, and to make and accept
  connections with paired devices.</string>
```

Code 13: BLUETOOTH's label and description.

5.3.1.2 Permission to GID Mapping

Inside the /data/etc/platform.xml file, BLUETOOTH and all low-level permissions are mapped to their GIDs. An application granted with one of these permissions will not only be running with its own UID attached to it, but also with the mapped GID. This allows its process to perform file system operations (i.e., read, write, and execute) which are restricted to that group.

```
<permission name="android.permission.BLUETOOTH" >
    <group gid="net_bt" />
</permission>
```

Code 14: Mapping BLUETOOTH to the net_bt group identifier.

All GIDs (and UIDs) are defined in /system/core/include/private/android_filesystem_config.h. Android 8.0 automatically generates lower case names based on these definitions (e.g., #define AID_NET_BT 3002 assign net_bt to UID or GID 3002). However, in prior versions, this connection needs to be done manually in the android_ids array.

```
...
#define AID_NET_BT_ADMIN  3001
#define AID_NET_BT        3002
#define AID_INET          3003
...

struct android_id_info {
    const char *name;
    unsigned aid;
};

static const struct android_id_info android_ids[] = {
    ...
    { "net_bt_admin",  AID_NET_BT_ADMIN, },
    { "net_bt",        AID_NET_BT, },
    { "inet",          AID_INET, },
    ...
};
```

Code 15: The declaration of the net_bt GID.

5.3.2 Access Permission

To restrict the access to BLUETOOTH's resources, Android changes the access permissions (which are not to be confused with our normal/dangerous, low-level/high-level permissions) of their related files. One way to do this is through init files, which are responsible for the system's initial setup. For BLUETOOTH in particular, this is done by /devices/<device brand>/<device name>/init.<device name>.rc, which provides device-specific initialization instructions (e.g., in our case, we used a LG's Nexus 5X in our tests, whose device name is bullhead, thus the file is `/devices/lge/bullhead/init.bullhead.rc`).

```
chown bluetooth net_bt /sys/class/rfkill/rfkill0/device/extldo
chown bluetooth net_bt /sys/class/rfkill/rfkill0/state
chown bluetooth net_bt /dev/ttyHS0

chown bluetooth net_bt /sys/class/rfkill/rfkill0/type

chmod 0660 /sys/class/rfkill/rfkill0/device/extldo
chmod 0660 /sys/class/rfkill/rfkill0/state
chmod 0660 /dev/ttyHS0
```

Code 16: Changing ownership and access permissions to `BLUETOOTH`'s related resources upon initialization (the lines were reordered for better visualization).

The `chmod 0660` command restricts read and write operations to only the owner of the files and his group. In other words, only processes with the `bluetooth` UID or the `net_bt` GID assigned to them can read or write the files contained in the first three files listed in code 16, which are located within the Android device's file system, instead of its source files.

The `/sys/class/rfkill` folder contains files responsible for controlling the system's radio transmitters. `/rfkill0/type`, for example, explicitly tells us the `rfkill0` folder is responsible for Bluetooth (in bullhead specifically) and `/rfkill0/state` determines whether it is enabled or not. Most importantly, `/dev/ttyHS0` is the file which serves as an interface for the Bluetooth device's driver, hence allowing software to communicate with it.

Therefore, new low-level permissions can be created whenever we want to restrict the access to entire files or even folders.

5.3.3 High-Level Permissions

Unlike low-level permissions, high-level ones protect portions of code, e.g., classes and methods. The steps to create and enforce permission of this kind are very similar to those followed when using a dangerous permissions inside an application. Let us see how this is done with `ACCESS_FINE_LOCATION` and `ACCESS_COARSE_LOCATION`, since we are already familiar with them.

5.3.3.1 Manifest Declaration

Exactly the same as BLUETOOTH, both permissions need to be declared in Android's own manifest, with the same attribute types.

```
<permission android:name="android.permission.ACCESS_FINE_LOCATION"
    android:permissionGroup="android.permission-group.LOCATION"
    android:label="@string/permlab_accessFineLocation"
    android:description="@string/permdesc_accessFineLocation"
    android:protectionLevel="dangerous" />

<permission android:name="android.permission.ACCESS_COARSE_LOCATION"
    android:permissionGroup="android.permission-group.LOCATION"
    android:label="@string/permlab_accessCoarseLocation"
    android:description="@string/permdesc_accessCoarseLocation"
    android:protectionLevel="dangerous" />
```

Code 17: The declaration of ACCESS_FINE_LOCATION and ACCESS_COARSE_LOCATION.

5.3.3.2 Permission Enforcement

In the previous section, we've seen that additional restrictions might need to be enforced in multiple classes and methods, since more than one of them might offer the same service. The same is valid for permissions. In LocationManager, for instance, ACCESS_FINE_LOCATION and ACCESS_COARSE_LOCATION are checked inside its getLastLocation() method, which return the last location data obtained by a registered provider.

First, inside the method, a "resolution level" is assigned to the application depending on which permission is granted. It is an integer that determines whether the app receives the location data and how accurate it is.

```java
// Location resolution level: no location data whatsoever
 private static final int RESOLUTION_LEVEL_NONE = 0;
 // Location resolution level: coarse location data only
 private static final int RESOLUTION_LEVEL_COARSE = 1;
 // Location resolution level: fine location data
 private static final int RESOLUTION_LEVEL_FINE = 2;

...

private int getAllowedResolutionLevel(int pid, int uid) {
    if (mContext.checkPermission(android.Manifest.permission.ACCESS_FINE_LOCATION,
            pid, uid) == PackageManager.PERMISSION_GRANTED) {
        return RESOLUTION_LEVEL_FINE;
    } else if (mContext.checkPermission(android.Manifest.permission.ACCESS_COARSE_LOCATION,
            pid, uid) == PackageManager.PERMISSION_GRANTED) {
        return RESOLUTION_LEVEL_COARSE;
    } else {
        return RESOLUTION_LEVEL_NONE;
    }
}
```

Code 18: A method inside /frameworks/base/services/core/java/com/android/server/LocationManagerService.java which returns the application's resolution level.

Then, if the resolution level is RESOLUTION_LEVEL_FINE, the location data obtained by the provider is returned unaltered, accurate; if it is RESOLUTION_LEVEL_COARSE, the data is time restricted, which means repeated method calls will return the same location (even if the provider obtained a new one), and obfuscated by the aforementioned LocationFudger; else, if it is RESOLUTION_LEVEL_NONE, no value is returned.

High-level permissions are, therefore, used when a more fine-grained enforcement is desired (instead of restricting entire files or folders, like low-level permissions do).

5.3.4 Examples

5.3.4.1 Low-Level Permission

Android devices have an internal storage, which by default is private (meaning no application, nor user, can access another's internal storage), and a shared external storage. Applications only need either the READ_EXTERNAL_STORAGE or WRITE_EXTERNAL_STORAGE dangerous permissions to have full access to the latter which, among other things, includes the device's saved pictures and videos.

Since low-level permissions can protect entire files and folders we could, instead of having permissions for the entire external storage, have particular ones for each folder containing a type of file. It would decentralize the access power of applications granted with the current external storage permissions.

This approach is certainly not the best one, specially since flexibility is a valued quality in this study. However, what can be accomplished with this kind of permission is rather limited due to its robustness.

5.3.4.2 High-Level Permission

In section 5.1, we've attempted to improve the issue of how information is shared by turning communication-related permissions (i.e., `INTERNET`, `BLUETOOTH`, `BLUETOOTH_ADMIN`, and `NFC`) into dangerous permissions. Another approach would be to still allow applications to automatically obtain them, but to restrict their ability to send and receive data behind new dangerous permissions, e.g., `SEND_DATA` and `RECEIVE_DATA`.

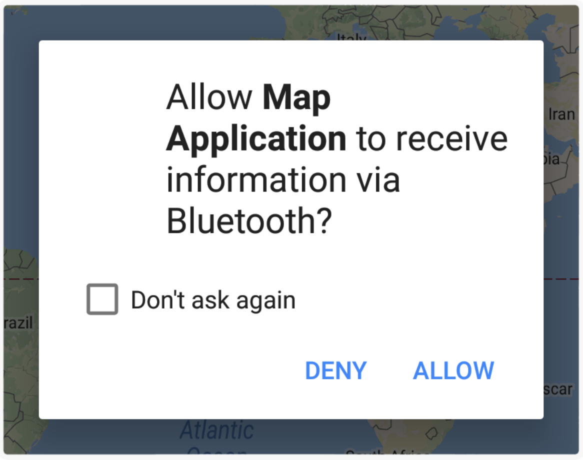
We've focused on restricting applications from receiving information through Bluetooth. At the risk of repeating ourselves, we've began by declaring the `RECEIVE_DATA` permission and adding the appropriate label and description string resources.

```xml
<string name="permlab_receiveData">Bluetooth Data</string>
<string name="permdesc_receiveData">receive information
  via Bluetooth</string>
```

Code 19: The label and description of the `RECEIVE_DATA` permission in `strings.xml`.

```xml
<permission android:name="android.permission.RECEIVE_DATA"
android:label="@string/permlab_receiveData"
android:description="@string/permdesc_receiveData"
android:protectionLevel="dangerous" />
```

Code 20: The declaration of the `RECEIVE_DATA` permission in `AndroidManifest.xml`.



Allow **Map Application** to receive information via Bluetooth?

Figure 5.2: An application requesting the new `RECEIVE_DATA` permission.

With Bluetooth, incoming data is handled by the `InputStream` class and all classes extending it. To enforce our newly created permission, we could modify these classes' `read()` methods, which read data from an input stream, effectively receiving it from somewhere else. However, they are native to Java, hence changing them is not a recommended procedure. Therefore, we've instead opted to protect the `BluetoothSocket.getInputStream()` method instead, which returns the input stream associated with a Bluetooth connection.

```
public InputStream getInputStream() throws IOException {
    return mInputStream;
}
```

Code 21: The original `getInputStream()` method, where `mInputStream` is the associated input stream.

```
import android.content.Context;

...

public InputStream getInputStream(Context context) throws IOException {
    if (context.checkSelfPermission(android.Manifest.permission.RECEIVE_DATA)
            == PackageManager.PERMISSION_GRANTED)
        return mInputStream;
    else
        throw new SecurityException("Missing RECEIVE_DATA permission");
}
```

Code 22: The modified `getInputStream()` method.

Modifications like this can, for example, prevent the scenario depicted by figure 4.2 from happening. They, again, stop applications from freely using the Android smartphone as an access point, an intermediary which sends and receives data from external devices, specially IoT ones.

Additional permissions can also be created to protect the smartphone's sensor which are not currently associated with any permission (e.g., the motion and light sensors). With the downside of bothering users with more requests, applications would be prevented from creatively obtaining sensitive information, such as the device's PIN code, through them.

Chapter 6

Awareness

All the measures tested and presented in the previous chapter are very intuitive. Creating new permissions increase the scope of reach of the permissions system, while adding restrictions to existing ones increase its flexibility. At the very least, they fulfill the task of better addressing the users' privacy concerns by expanding their options. However, their—and the entire permission system's—capability to upkeep privacy is entirely reliant on the dubious premise that users are aware of the dangers of granting permissions to applications.

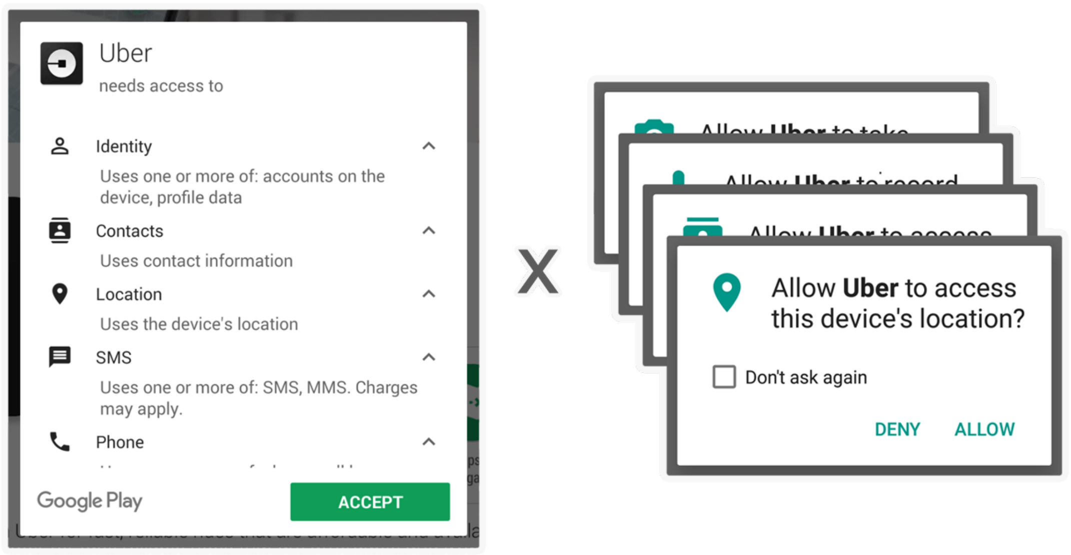
This absence of risk awareness is among the greatest threats not only to permissions, but privacy as a whole. In this chapter, we will discuss the two main problems encountered that cause this generalized issue and how they affect the permission system's efficiency.

6.1 No-Reading Problem

The first problem is the population's lack of attention to contracts, regardless of subject or importance. Ian Ayres and Alan Schwartz of Yale Law School call it the "no-reading problem" [10]. It was previously restricted to printed documents, such as cellphone contracts, insurance policies, and mortgage agreements, but quickly encompassed digital contracts as well. They give us the example of PC Pitstop's users, who took four months to notice and claim a reward of a thousand dollars which was announced in the website's end-user license agreement (EULA).

In Android, the no-reading problem affect the users' ability to read permission request prompts. According to Rainer Böhme and Stefan Köpsell, "Ubiquitous EULAs have trained even privacy-concerned users to click on 'accept' whenever they face an interception that reminds them of a EULA." [11]. Therefore, adding new permissions can be extremely inefficient if users mindlessly grant them to applications.

Android 6.0's runtime permissions were certainly an attempt to tackle this problem. By dividing the large, all-in-one permission request into multiple smaller ones, their resemblance to EULAs was diminished. Hence, it is possible the users' attention to permissions reached acceptable levels with this change. However, the opposite is just as, if not more, likely because this increase in number of interceptions (requests) might've desensitized users even further. Unfortunately no reliable research was found on the impact of Android 6.0's new permission model.


Uber
needs access to
Identity
Uses one or more of: accounts on the device, profile data
Contacts
Uses contact information
Location
Uses the device's location
SMS
Uses one or more of: SMS, MMS. Charges may apply.
Phone
Google Play
ACCEPT

X

Allow Uber to take
Allow Uber to record
Allow Uber to access
Allow **Uber** to access this device's location?
Don't ask again
DENY ALLOW

Figure 6.1: Uber's pre-6.0, all-in-one permission request versus its runtime requests.

Since the no-reading problem stems from a deeply-seated disposition, albeit a nurtured one, there is a strong argument to be made that it is not within Android's capabilities to solve it. That is not to say the operating system doesn't have the responsibility to mitigate the problem within its confines. However, such a big shift in behaviour would require a much greater effort, an effort that the permission system cannot hope to take part alone.

6.2 Misguided Expectations

The second problem, which causes the lack of risk awareness, is the population's misguided expectations. According to Nissenbaum, many privacy violations occur when personal information is used in unexpected ways [12]. Uber's users, for example, most likely gave the application their location because they didn't expect it to abuse this privilege, they assumed a certain level of confidentiality was warranted; nevertheless, as we've previously seen, the app did abuse it.

With permissions, users' expectations are misguided by two things: the request's lack of information and the context it is made in. When an application requests to "take pictures and record videos", most users don't expect it to be able to do so at any given time, much less actually do it. They don't have the necessary experience or knowledge. Also, because said request is usually made when the users themselves desires to take pictures or record videos, they further expect it to only use the device's camera when they do. Therefore, even attentive users can fall victims to an app's exploitation.

IoT worsen this problem even further. It increases the number of connected devices and, thus, the complexity of the whole system. As a consequence, the possibilities of how personal information can be obtained are drastically escalated, leaving users way more open to the unexpected.

At first glance, the obvious solution would be to increase the amount of information contained in a request; users reading it would be able to better assess the risks and, thus, tread more carefully. However, it's also possible this measure could either be too effective, making users overly wary of applications, reducing Android's appeal to both, or have the opposite effect: more detailed requests could revert their resemblance to contracts like EULAs, decreasing the users attention to them. Sadly, a better assessment of possible solutions would require a more detailed research on the subject of human comportment, else we would only be dealing with speculation.

Chapter 7

Conclusion

Throughout this work, we learned what permissions are and how applications use them. We established three main issues with Android's current permission system, based on Alan Westin's definition of privacy: permissions don't allow users to discern when personal information is being obtained by an application, how (by which means) it is being obtained, and to what extent. With these issues in mind, we studied improvements that could be made to the system, all of which were relatively straightforward due to how well-structured its source files and code were (with the challenge rather being to locate the appropriate places to make their respective changes).

However, in the process, we noticed there was a greater underlying problem that hindered the permission system's effectiveness: the users' lack of risk awareness. All the improvements proposed relied on the users' ability to recognize how hazardous granting a permission can be. A premise that is generally not true, or at least wasn't before Android 6.0's new permission model.

Finally, we better understood this problem and its causes within and without Android, and acknowledged that, in order to solve it, much more drastic changes would need to be made. Changes that are, unfortunately, not in the scope of this study.

Unfortunately, not a lot o emphasis was given to IoT, even though it was our initial intention to. Due to time constraints, we focused towards permissions inside applications and Android itself, straying further from IoT's bigger picture. We judged it was more natural to our (and the reader's) learning experience to do so, as it would've certainly felt rushed otherwise.

Bibliography

[1] G. Orwell. *1984*. Houghton Mifflin Harcourt, 1983. 1

[2] Samuel D Warren and Louis D Brandeis. The right to privacy. *Harvard law review*, pages 193–220, 1890. 3

[3] Alan F Westin. Privacy and freedom. *Washington and Lee Law Review*, 25(1):166, 1968. 3

[4] H Jeff Smith, Sandra J Milberg, and Sandra J Burke. Information privacy: measuring individuals' concerns about organizational practices. *MIS quarterly*, pages 167–196, 1996. 3

[5] Steven Bellman, Eric J Johnson, Stephen J Kobrin, and Gerald L Lohse. International differences in information privacy concerns: A global survey of consumers. *The Information Society*, 20(5):313–324, 2004. 3

[6] Patrick Kelley, Sunny Consolvo, Lorrie Cranor, Jaeyeon Jung, Norman Sadeh, and David Wetherall. A conundrum of permissions: installing applications on an android smartphone. *Financial cryptography and data security*, pages 68–79, 2012. 6

[7] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the eighth symposium on usable privacy and security*, page 3. ACM, 2012. 6

[8] Newcastle University. How criminals can steal your pin by tracking the motion of your phone, 2017. http://www.ncl.ac.uk/press/news/2017/04/sensors/. 11

[9] Spangenberg v. uber technologies, inc., 2016. 12

[10] Ian Ayres and Alan Schwartz. The no-reading problem in consumer contract law. *Stan. L. Rev.*, 66:545, 2014. 30

[11] David Berreby. Click to agree with what? no one reads terms of service, studies confirm, 2017. https://www.theguardian.com/technology/2017/mar/03/terms-of-service-online-contracts-fine-print. 30

[12] Helen Nissenbaum. Privacy as contextual integrity. *Wash. L. Rev.*, 79:119, 2004. 31