SAPIENZA
UNIVERSITÀ DI ROMA

Data Leak Detection in Mobile Applications

Confidentiality of Personal Data

Dipartimento di Ingegneria Informatica, Automatica e Gestionale
Engineering in Computer Science

Iommazzo Nicola
ID number 1693395

Advisor
Prof. Leonardo Querzoni

Co-Advisors
Pierluigi Pierini
Paolo Piccolo

Academic Year 2022/2023

Data Leak Detection in Mobile Applications
Master thesis. Sapienza University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: iommazzo.1693395@studenti.uniroma1.it

Acknowledgments

Acknowledge n1
Acknowledge n2

TODO:

- *Abstract*
- *App case: Radarbot*
- *App case: Waze*
- *App case: GoogleMaps*
- *Conclusion*

Abstract

Briefly describe the whole content of the thesis.

Contents

Chapter 1

Introduction

Mobile applications are now part of our daily life. Everyone of us make use of different applications during their daily routine. There are applications for literally every kind of necessity: we would like to know what the weather will be like in the next days in our city, we would like to keep track of our fitness training that every morning we do in the neighbourhood, or we would like to know how the traffic is on the way back to our home. Using these applications is becoming a real habit, and like any other habit we do not ask ourselves anymore why or how we are doing it. Going through the years much more often we can hear about *data breaches* or *data leaks* related to applications. The whole amount of data that users has put in their online account can be exposed to risks of this kind. Nowadays people tend to use applications without even realise the amount of informations they are delivering to the network: profile pictures, email addresses, phone numbers. All of these are private informations, and a disclosure of them can be really embarassing for both the user and the company offering the service.
Mobile applications handle private data of millions of users, therefore it is mandatory checking the robustness and any kind of protection implemented in the application, in order to make it resilient to possible leaks of user informations.

Main topic of this thesis is to investigate multiple mobile applications with a special focus on the personal informations they handle. A critical eye is directed towards any outgoing application request, looking for any vulnerability in the code or in the communication protocol adopted. If by any chance a vulnerability is discovered, then it is mandatory to check if it can be exploited in order to obtain personal informations.

The structure of the thesis proceeds by chapters. This introductive chapter aims to describe the goals of the study and the methodologies used while conducting the investigation on every different mobile application. The second chapter is dedicated to the explaination of the concepts and the technologies met during the work. The third chapter deals with the whole testing environment in which every application has been analyzed, specifying the potentiality of each tool. The fourth chapter explains, application by application, how the investigation has been practically conducted, the data exposed, the vulnerabilities found,

and if any, how they can be exploited to obtain private informations. The last chapter is a summary of the whole study, reporting the results of the investigation process in every application.

Personal Data

At the really start of this thesis I would like to define the concept of **Personal Data**. There are multiple definitions and laws defining the scope of this kind of data. As stated by the European Commission [2]:

> *Personal data is any information that relates to an identified or identifiable living individual. Different pieces of information, which collected together can lead to the identification of a particular person, also contribute personal data.*
>
> *Personal data that has been de-identified, encrypted or pseudonymised but can be used to re-identify a person remains personal data and falls within the scope of the GDPR. [...]*
>
> *The GDPR protects personal data regardless of the technology used for processing that data - it is technology neutral and applies to both automated and manual processing. [...]*

Practical examples of informations falling in the category of personal data are - name and surname, address, phone number, date of birth, but also photographs, ip addresses, location informations.

Dealing with mobile applications, personal informations are often typed in directly by the user, most likely at the moment of the creation of a new user profile, but not only. Some personal data might be shared automatically during the use of such application, like ip address or advertising phone identifier.
These information not necessarily are mandatory to access the services offered by the application. Personal data might contribute to some advertising service bundled within the application to keep track of the user data, or these data might be sent to some logging service implemented in the application in order to retrieve statistics on the users utilizing that application.

1.1 Application choice

The work was conducted by examining applications running on Android operative system. Different mobile applications have been analyzed belonging to three different area of interests:

- Weather: General applications *without* user authentication.
- Health & Fitness: Applications *with* user authentication and *some* interaction between users.

- Maps & Navigation: Sophisticated applications *with* user authentication and *continuous* interactions with server or other users.

The order in which the applications have been investigated has been decided basing on an increasing complexity and notoriety of them. Starting from weather forecast applications that generally do not require any kind of user authentication, going through fitness applications that let the user customize and share fitness routines to other users, and finally analyzing applications that manage geographical informations to retrieve real time traffic data.

1.2 Goals

Goal of the study in this thesis is to check if software features expose it to the risk of providing more information than planned. While investigating the behaviour of an application there are two questions I have tried to answer.
The first one is *"Are there any private informations disclosed by the application, and if yes, in which way these informations are protected ?"*. Protecting the customers information is a duty of every company while developing any of its services. It is mandatory ensuring that private and sensitive informations are safely stored.
After having anwered to that, then the second one *"There is any vulnerability that can be exploited in order to obtain those private informations ?"*. Of course a vulnerability is not present on purpose in an application, the human error is always possible, and it is important be aware of this possibility.

A mobile application might share private informations in different moments of the execution, just while opened, while using it on top of the screen or while running in background. Goal is therefore ensuring there is no leak of private data for the whole activity and at any moment of the application execution.

1.3 Methodologies

Since the goal is to detect any personal data leak while using the application, dynamic analysis is the main method used to investigate applications behaviour. In particular each mobile application has been inspected starting from the **network traffic analysis**, generated by the application at runtime.
Every action carried out by the user in the context of an application will generate some request to a server. Notice that mobile applications do not communicate only with the server offering that service. Indeed in the implementation of a mobile application there are different activities, each one provide a different service - for example advertisement services, logging services, push notification services, and so on. Every request is therefore generated by the application and might potentially include private informations.

Where the network protocol analysis offered a spotlight for some in-depth study, **static analysis** has been adopted going through an examination of the code of the application.

Performing static analysis on Android applications is really dispersive and time consuming. An Android Package file (APK) can contain hundreds of thousands of Java classes. The majority of the compiled applications is stripped from the symbols, so methods and classes names cannot be directly found in the source code. Most of the time a simple obfuscation method just like the described one, combined with the high number of classes, it is sufficient to hide something very interesting, as the method used to compute the *Authorization header* in an HTTP request is.

To know more about the static and dynamic analysis tools I used in the practice, go to Chapter 3.

1.4 Collaboration with IPS

The whole study described in this thesis, has been supported by the company *IPS*[1].

IPS was founded in 2000 as a company specialized in the development of solutions and provision of services for the Cyber Security sector and is today an important national reality which offers solutions and products with proprietary technology in the fields of Communication Security and Electronic Surveillance.

The collaboration with IPS and the technological exchange that took place during the research followed two principles:

- **Cyber defense**: Computer security is a fundamental asset for the business of any company and to defend it is necessary to set up an efficient cyber security system. IPS is a global provider of Cyber Intelligence solutions with more than 30 years of experience in the high-tech market.
  IPS continually revises and refines verification procedures of industrial assets and infrastructures as well as tools used by staff; both for its own protection and as a service offered to customers.
  The continuous analysis of apps, software and vulnerabilities, as done in the specific work of this thesis, is thus part of the core business of IPS.

- **Investigation**: As security is becoming a major concern with threats coming from both the real world and the virtual Internet world, Law Enforcement Agencies see the complexity of investigations increasing and ask for more and more sophisticated technology to provide up to date monitoring capabilities for surveillance of both the real world, the traditional communication services and the Internet. IPS provides LEAs with strategic and tactical solutions for Communication Intelligence needs. Deep knowledge of leaks and data exposure, as done in the specific work of this thesis, are the basis of a technologically advanced investigation work capable of bringing results in the most complex situations.

Chapter 2

Fundamentals

This chapter is dedicated to illustrate some fundamental knowledge in order to fully understand the work I have done while investigating specific applications. Some of them are technologies arised in last ten years, improving not only the efficiency but also the stability and portability of the standard development of mobile applications.
Generally Android mobile applications are client implementations for specific services. The scenario is therefore the classic client-server communication model where each client interacts with one or more server.

First of all client and server must establish a connection between them, hopefully a *secure connection*. This is the first topis explained in this section. Client and server, that means mobile application and application server, agree to find a secure way to communicate each other.

Once the connection is established there are lots of ways in which an application can send information. This is the problem of the data serialization. Informations can be represented using different Interface Description Languages (IDL). Starting from the classic JSON or XML format, passing through binary. Each representation has advantages and disadvantages. In particular there is a section dealing with an IDL that is widely used in mobile application, that is *Protocol Buffer*.

After having described the interation of a standard mobile application, there is a section explaining the concept of RPC, and in particular the recent developed *gRPC* framework. As I said most of the mobile applications reflect the client-server paradigm. It is modern framework able to delegate some procedure in the client to the server, improving performance and portability of the application service.

Finally the last section of this chapter deals with a new network procol called *QUIC*, a modern evolution of the HTTP/2 protocol used by most of the recent applications.

2.1 HTTPS: HTTP over TLS

As stated in the online documentation for Android Developers [3] it is a good practice to protect applications data using the **Transport Layer Security (TLS)** along with the

standard HTTP protocol.
TLS is a protocol designed to provide communications security over a computer network. It provides *confidentiality*, *authentication* and *integrity* meaning that: data is encrypted for third parties, ensure that both parties actually are who they pretend to be, ensure that data is not modified in the transit.

A server (or web application) that wants to use TLS must have installed an **SSL certificate** on the machine. This certificate is released by a **Certificate Authority (CA)** for a specific domain, and contains important informations on the owner of that domain together with the public key of the server.

Every time a client wants to establish a connection with a server using TLS, the TLS Handshake procedure is issued.

There are also some mitigations Android mobile application developer can use in order to limitate authority checks

2.1.1 SSL/TLS Handshake

During the TLS handshake both mobile application and server exchange informations to set up a bidirectional encrypted connection. For this reason TLS version (1.0, 1.2, 1.3, etc) and cryptographic suite are agreed. Moreover the client verifies the authenticity of the server basing on its certificate. Finally both the endpoints will generate session keys to encrypt and decrypt the messages they will exchange each other. Once data are encrypted, they are signed with a Message Authentication Code (MAC), so that the receiver can verify the integrity of the data sent over the network.

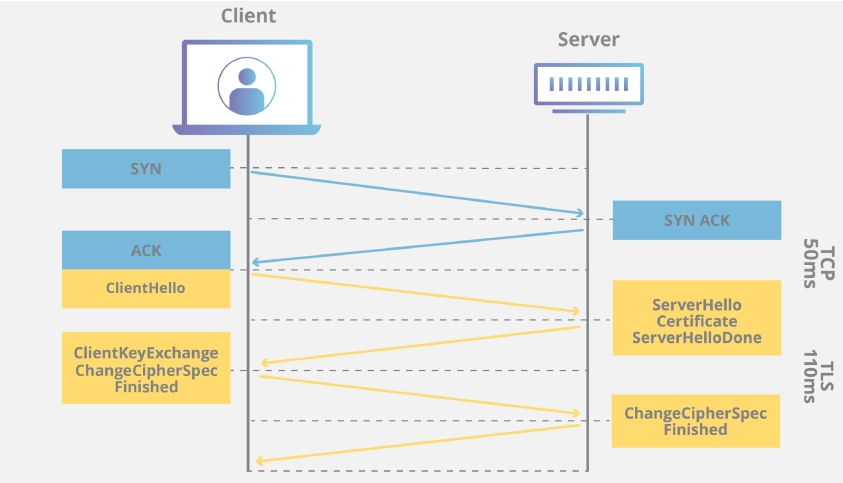


Figure 2.1. TCP Handshake and TLS Handshake communication flow needed to set up a secure connection. [4]

From the performance point of view, the whole process of the TLS handshake needs both the endpoints to communicate before the data are actually transmitted over the network. In particular in the TLS versions < 1.3, two round trip time are needed to set up the whole connection *(110ms)*. This amount of time added to the standard TCP three-way-handshake *(50ms)* to establish the session between client and server, results in average in less then *0.2*

seconds spent only to set up a secure connection without actually sending any application data yet.
As I said TLS makes possible to exchange information over a non-secure channel by providing confidentiality, authentication and integrity.

2.1.2 Certificate Verification

A procedure I would like to stress is the certificate validation routine that happens at the really start of every TLS handshake. At the moment the client wants to establish a connection with the server, it needs the certificate to check the server is the actual endpoint the client want to talk with. The first thing the client does is to check that the certificate showed by the server is not expired, and that the server domain name matches with the one reported in the certificate. The next step performed by the client is to verify that the certificate has been signed by the certificate authority that authorized it. This procedure is done by verifying that the issuer's CA name matches the owner's CA name at the above level, and using the owner's CA signature and public key to verify the certificate is properly signed. The process is repeated until a CA trusted by the client is met, usually a root CA.

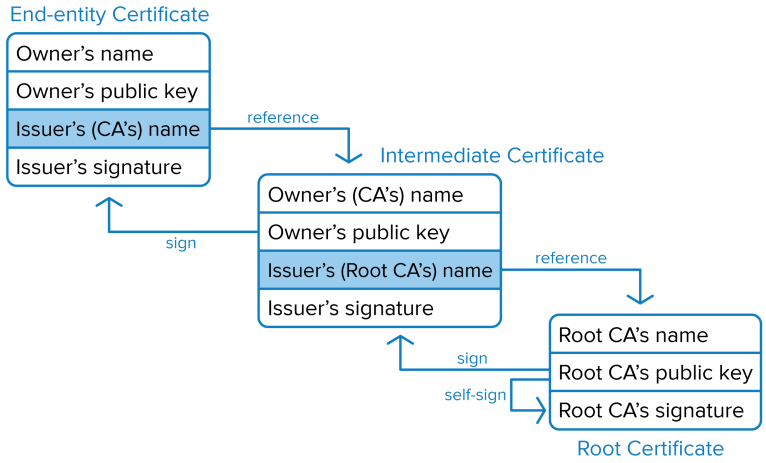


Figure 2.2. TLS Certificate verification chain. [5]

Usually certificates are bought by companies that wants to implement a secure connection with their clients. In this way the certificate obtained by the server is inserted in the tree of certificates, and every client can verify its authenticity. Note also that every machine can generate a self-signed certificate. What is indeed important is checking if the server certificate is linkable to a trusted Certificate Authority.

The Android operative system handles by itself a list of trusted certificates. These certificates belong to two categories: system certificates and user certificates.
System certificates are used by default to verify all other certificates. So if we are using an application trying to connect to an application service, if the certificate obtained by the web

service is traceable to one certificate already present in the system store, then the certificate is accepted and the connection is established. Notice that since the release of Android 7 Nougat, it is impossible to install any CA at system level without having root privileges on the device.
On the other way *user certificates* are installable even at user level, and are used to verify certificates when directly expressed by the application.

2.1.3 Man-in-The-Middle

Man-In-The-Middle (MITM) is an attack able to disrupt the security properties delivered by the TLS protocol. As the name suggests the attacker positions himself between the server and the client. The attacker acts like a proxy server intercepting every connection incoming from the client, pretending to be the real server, and forwarding them to the real server, pretending to be the real client.

On one side, at the moment of the TLS handshake, the client verifies the identity of the server, so the attacker needs to reproduce the handshake with the client in order to let it trust the attacker. As described in the last section, the authenticity check is performed through the exhibition of the server certificate. In this case the client needs to trust the certificate of the attacker. After having checked the identity of the server, the TLS handshake can continue following the standard procedure: client and attacker agree on a pre-master shared key and the encrypted communication will take place.

On the other side the attacker needs to act as the real client for the server, so that every connection intercepted will be redirected to the server, and every response obtained will be delivered to the client. This is made possible by starting a new connection between attacker and server. The attacker will authenticate the server by checking its certificate and completing the standard TLS handshake with it.
At this point the attacker has performed two different TLS handshakes, one with the client faking to be the server, and one with the server faking to be the client. The man in the middle is now able to read messages sent or received from the client and forward or deliver them to the server.

A man-in-the-middle attack is what made possible the analysis of the communication protocol of every application I investigated. After having *placed* ourselves inbetween client and server we will be able to understand what exactly is the protocol adopted by an android application and analyze each message looking for a possible leak detection of users personal data. For more specifics on how I performed a MitM attack for my case studies, read the chapter *Testing Environments*, sections *HttpToolkit*[Section 3.2.1] and *BurpSuite*[Section 3.2.2].

2.1.4 SSL Certificate Pinning

If the MitM attack results in the loss of authenticity in an HTTPS communication, the **SSL Certificate Pinning** is an hardening process that can be adopted in order to deliver an additional layer of security to the software. Pinning certificates or public keys is a

countermeasure that allow web servers to control the risk of man-in-the-middle attacks or CA compromise.

The whole process of certificate verification described in the previous section is skipped, the application will indeed only validate certificates or public keys that are *pinned* to the application. A MitM attacker will not be able to intercept messages anymore. The connection will be blocked right in the middle of client's and attacker's TLS handshake. The application implementing a certificate pinning method in fact will not trust anymore the attacker's certificate, and the connection will be dropped by the attacker's proxy.

There are two ways of implementing this hardening technique. The first one is by *preloading* the public key or certificate in the client at development time, so that every fake server trying to connect to the client is rejected during the TLS handshake phase. The second one happens at runtime by installing the *pin* in the client once upon the connection is established by using the HTTP Public Key Pinning (HPKP) header, indicating the public key and the max age of validity for that specific pin. This second approach has been adopted by Chrome and Firefox for their browsers, but after some years new security issues has been discovered about it. Indeed in case of bad implementation (for example a pin of an incorrect public key) could bring to denial of service, but even in case of compromised server the consequences can be devastating for a longer time if HPKP is implemented, a bad pin might be established from an attacker that had the control of the server (HPKP Suicide [7]). For this reason the public key pinning is now highly discouraged for web browsers.

In Android application SSL certificate pinning is a solution still able to protect from MitM attacks. The implementation is generally done by pinning a specific certificate inside the application logic at development time. There are multiple ways to implement a certificate pinning in Android as explained in [8]. Among them two approaches can be identified:

1. basing on *Network Security Configuration*: A pretty simple certificate pinning implementation made possible by specifying a *<domain-config> tag* in the network security configuration file present in the application. A *pin-set* of *pin* can be specified for the whole application.

2. basing on the *network library*: Many different network libraries exist for Android application (i.e. *OkHttp*, *Retrofit*, *Volley*, and so on). Each one of them provide methods to skip the standard certificate validation procedure and implement the certificate pinning procedure.

2.1.5 Certificate Transparency

Certificate Transparency (CT) is a security standard adopted by browsers for monitoring and logging discrepancies in the certificate verification procedure. The whole system is manteined by a set of certificate logs handled by Certification Authorities and many browsers. These logs assume the form of *Merkle trees* (also noticed as *hash tree*), they are publicly verifiable, in append-only mode. The goal of this standard is to keep track of all valid certificates ever issued for a specific domain.

This is the scenario: a domain owner implementing a web service requests a new certificate from a CA. Before releasing the new certificate, it submits the *hash digest* of that certificate to one or more certificate logs, receiving a *Signed Certificate Timestamp (SCT)* signed by the log provider. Once signed, the CA can now send back the certificate with embedded the SCT to the domain owner asking for the certificate. Every HTTPS connection to that web service will be served using that certificate. When a user wants to use the web service, browsers or mobile apps will query the trusted certificate log to find a record matching with the SCT of the certificate received by the web service.

In Android operative system, an example of application implementing the Certification Transparency mechanism is the Chrome browser. As we said above (see Certificate Verification [Section 2.1.2]) Android certificates can be placed at system level or user level. Every application will use by default the system store to verify others certificates, and so does Chrome, if not specified differently. This application will choose whether to use the Certificate Transparency or not, basing on which store the certificate root is installed. When a certificate showed by any web service it is checked, if the root certificate met is present in the system certificates store then Chrome decides to also verify the Certificate Transparency requirement. On the other way if we are testing a web service showing a certificate that is inside the user certificates store, then the Certificate Transparency procedure is not applied.

2.2 Protocol Buffers

Protocol Buffers[9], also known as ProtoBuffers, are a language-neutral, platform-neutral extensible mechanism for serializing structured data. The development of this new mechanism started in the early 2001 from Google for private purposes. Then in the 2008 was publicly released as open-source data format. The design goal while developing Protocol Buffers was to provide a data serialization universal and faster then the standard XML. Nowadays it is widely used when an internaction between different architectures client-server is needed, that is the case for most of the Android application.

The messages exchanged are data structures described in a proto definition file *.proto*. After having compiled those definitions through *protoc* they can be imported in any project written in the most used programming languages. Protobuf version 2.0 was providing a code generator compatible with *C++*, *Java*, *C#* and *Python*. The version 3.0 extended the programming language compatibility also to *Go*, *Ruby*, *Objective-C*. Other languages are supported through third-party implementations (*C,JavaScript*, *Perl*, *PHP*, *Scala*).

To give an idea on how the mechanism works, let's say a client-server application would like to serialize data about a *Person*. In order to let both endpoint to know about the data structure *Person*, both of them require to integrate in their project that data structure. The definition of *Person* is done in a *.proto* file:

```
message Person {
  optional string name = 1;
  optional string email = 2;
  optional string phone = 3;
```

```
}
```

The proto file has to be done only once, and it is independent from the platform we will be using. To import the definition of the above *Person* data structure in our project we need to run the *protoc* compiler obtaining the ready-to-use structure.

Since the native language for Android application is Java, running *protoc* we are interested in the *.java* classes generated. For each message type defined in the *.proto* file will result a couple of *.java* files: one will contain the definition of the object itself, the other one is the Builder for that specific object type. The first one will be provided of the standard accessor's methods (in our case *hasName()*, *getName()*, and so on for each field), while the second one will have both getters and setters methods (*getName()*, *setName()*, and so on for each field). At this point it is needed to import those definitions in *.java* classes thanks to the Protocol Buffer API provided by the mechanism itself. This is a picture summarizing the whole procedure:

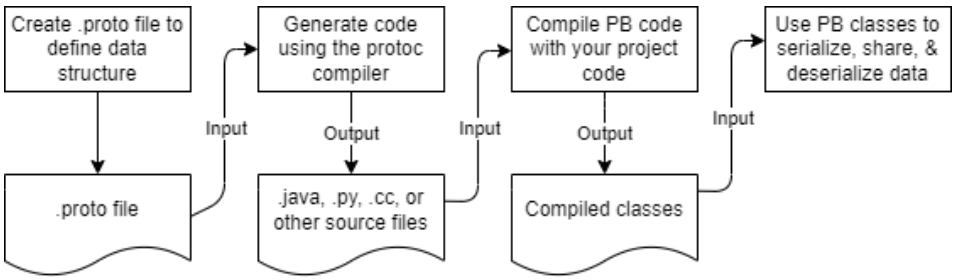


Figure 2.3. Procol Buffer workflow [9]

Each object already instantiated is immutable, it can only be modified through its respective Builder methods at the moment of its building. The procedure to instantiate a new *Person* object in the application is respectively:

1. Contruct a builder for the object.
2. Set any field to any acceptable value.
3. Call the builder's *build()* method.

In particular the step 2 is made possible through the mehods provided automatically in those *.java* files. Each one of them will modify the structure created at step 1, and it will return a new *Builder* object, such that it can be repeatedly modified and at the end instantiated through the *build()* method.

The above defined *Person* class, it is just an example. The Protocol Buffer mechanism is able to provide a platform-independant really complex classes with repeated fields, nested classes, constant values or even rpc services to define. At the same time the use of Protocol Buffer let the developer save time when developing the data structures. Indeed the *.proto* definition has to be done only once, and it is valid for every kind of platform that would use that type of structure.

2.3 gRPC

gRPC [10] is a cross-platform *Remote Procedure Call* (RPC) framework. The mechanism was initially created by Google for private purposes, but in 2016 it was publicly released as open-source. As a remote procedure call framework, it let the developer to define in an application some methods that will actually run on a different machine from the one running the application. Generally the idea behind any RPC system is to define a *service* able to execute a specific method in order to call it remotely from any another application.

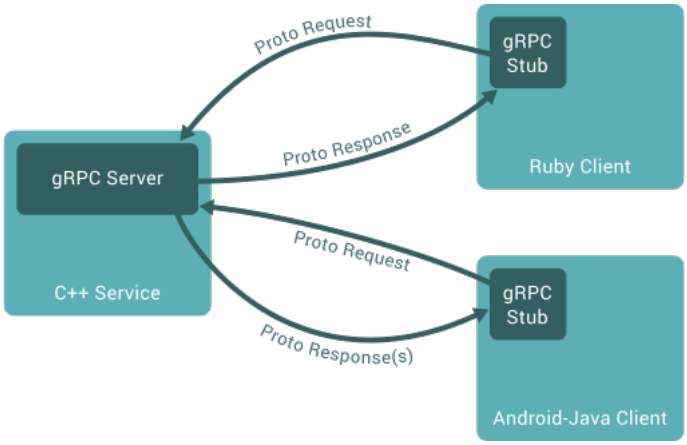


Figure 2.4. gRPC workflow [10]

What makes gRPC so powerful is the fact it is based on *HTTP/2* protocol for the transport layer, and uses *Protocol Buffers* (described in Section 2.2) as its Interface Description Language (IDL). The combination of these two mechanism let the gRPC system to provide an easy access and platform independant solution for the implementation of a RPC system. In a gRPC system, as any other RPC system, we can differentiate between *gRPC Server* and *gRPC Stub*, the part of the client implementing the gRPC framework. On the server side, the server implements the interface for the specific method it is able to execute, and handles the incoming client calls. On the client side, the gRPC Stub provides the same method as defined by the server. What really happens in the moment in which the client invoke the method is that:

1. The gRPC Stub serializes the parameters passed to the method using the Protocol Buffer mechanism. Then sends over an HTTP/2 connection the method request together with the serialized data.

2. The gRPC Server accepts the incoming request from the client and deserializes the data using the Protocol Buffer mechanism.

3. The gRPC Server will execute the method specified by the client.

4. Once the result is available, the gRPC Server sends over the HTTP/2 connection the response containing the return value for that specific method, serialized with the Protocol Buffer mechanism.

5. The gRPC Client deserializes the return value using the Protocol Buffer mechanism.

Even if different IDLs can be specified while defining the gRPC service, JSON for example, by default the gRPC framework uses Protocol Buffers. Moreover the framework provides features like *authentication* (through TLS or token-based authentication) and *bidirectional streaming*.

2.4 QUIC

QUIC is a general-purpose transport layer network protocol designed by Google. It was publicly announced in 2013 and after some years under the experimental state, in the recent 2021 became a standard as defined from the Internet Engineering Task Force (IETF) in *RFC 9000* [11].

The main goal of QUIC is to improve the performance of the connection-oriented web applications, that for years have been run basing on TCP as transport layer network protocol. QUIC establishes a number of multiplexed connections using UDP instead, allowing multiple streams of data to reach different endpoints independently. The same concept is possible in HTTP/2 with multiplexed connections, but what really suffers this last one is the the head-of-line-blocking delays related to the connection-oriented TCP which it relies on.

Briefly the *Head-of-Line blocking* (HOL blocking) is an issued generally affecting every packet that need to pass through a line. If the First-In-First-Out (FIFO) method is used, the longer the packet has arrived in the queue, the higher is the priority it has to be sent over the line. With the release of HTTP/2, this issue was partially resolved through the multiplexing of multiple connection, letting multiple HTTP requests running together but still an implementation over a single TCP connection. If one packet is being lost in the transmission than the whole connection will be waiting for the retransmission of that single packet. Now in the case of a client with few connections towards a bunch of server is waiting time is neglactable, but modern Android applications are implementing different services at the same application, using both standard HTTPS requests and gRPCs with an high number of structures to serialize through Protocol Buffer towards hundreds of endpoints. In case of a packet retransmittion the performance loss will be evident.

QUIC protocol addresses this issue by relying on *UDP* as transport layer protocol. UDP is a connectionless transport layer protocol without any implementation of packet orderding and retransmission, indeed every packet is forwarded to the endpoint individually, without any risk of HOL blocking problem. At the same time UDP does not provide any guarantee on the packets delivery making the whole connection unreliable. All these checks are in fact implemented in QUIC on top of UDP, while still taking advantage of the performances achieved by UDP. Every additional connection feature, like TLS encryption, is done in the same way on top of UDP.

The second goal of QUIC is to improve the connection latency between the two endpoints. Using UDP as transport layer protocol the TCP overhead needed for the connection management is already cutted off. Most of the applications will demand for TLS encryption requiring its own handshake, that would be summed up to the TCP three way handshake if using the HTTP/2 protocol. With QUIC there is no redundancy in those handshake phases. It is done only once in QUIC as showed below:

Moreover, as said above, UDP does not handle loss recovery, meaning that QUIC is also

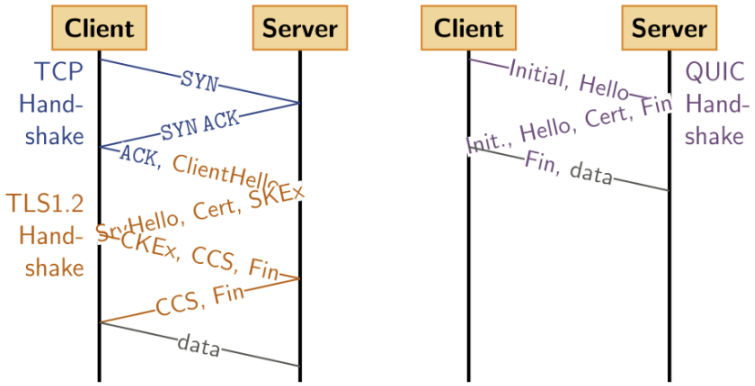


Figure 2.5. Comparison between TCP+HTTPS and QUIC handshakes

resposible for each separated controlled flow. Others improvement in terms of overall latency are adopted, for example packets are individually encrypted, so that can not result in a waiting for partial packet.

Another property of QUIC is the ability to mantain its performance during network-switch events that frequently happens in mobile devices, like the connection transition between WiFi hotspot and mobile network. In HTTP/2, based on TCP, every connection is timed-out one by one and then re-established on the new network, while in QUIC at every connection is assigned an identifier that is kept even if the source ip address changes.

Lastly, QUIC is completely handled at application level. There are numerous libraries available in order to implement this protocol in an application: *Cronet* developed by Google, *sn2-quic* developed by Amazon Web Services, *proxygen* by Facebook and so on.
For a practical example see the application case Google Maps in Section 4.5.

2.4.1 HTTP/3

In June 2022 the IETF published **HTTP/3** as the major version of the *HyperText Transfer Protocol* in *RFC 9114* [12]. Unlike the earlier versions of HTTP 1.1 and 2.0 based on TCP, HTTP/3 relies on QUIC, based on UDP.

HTTP/3 uses the same semantics of the previous versions (request methods, status codes, message fields) but it differs in the way it mantains the connection session. Most of the changes in HTTP/3 are in the way the protocol interfaces with the underlaying transport layer. The new version of HTTP results in lower latencies and higher performance in real-world usage, mostly due to the adoption of QUIC.

Chapter 3

Testing Environment

In this chapter is described the whole environment where I settled the investigation of each Android mobile application discussed in this thesis.

Core of the environment is the *Android Studio* framework, offering both an Integrated Development Environment (IDE) and testing features for Android mobile applications. Here I configured an emulated device where I could run the application cases I investigated.

Once installed the specific application I had to analyze the *network traffic* generated by itself. Generally it is rare a single tool is perfect for all the applications to analyze, this is the reason why I used different tools to inspect the network traffic generated by an application, instead of just sticking with one tool. Starting from the standard-de-facto in packet inspection *Wireshark* program, ending to tools acting like proxy servers between the client application and the server. Some of these tools are able to automatically install and configure a man (MitM attack [2.1.3]) in the middle of the communication, others might need some manually configuration.
The whole amount of informations exchanged between application and server is hidden somewhere in the communication. It might be more or less difficult to find them, but all the informations are there, in those packets.

Other tools revealed to be aswell useful while investigating Android applications. Dynamic instrumentations tools are softwares able to run scripts from the inside of the applicaiton, as they were part of it. At the same time, static analysis tools provided informations on class and libraries used by the application, simply basing on source code of the application itself.

3.1 Android Studio

Android Studio is the center of the testing environment. Other than a complex IDE software, it provides a complete testing suite for Android mobile applications. Once installed the software, it enables the access to the Android Device Manager tool, where the user can create an emulator for a specific Android device choosing among phones, tablets, wearOS or TVs. In this section I will explain how I prepared the android virtual device in order to obtain a working environment that made possible the application investigation.

3.1.1 Android Virtual Device (AVD)

For my purpose study case I created an *Android Virtual Device* (AVD) phone with the same properties of the *Pixel 3a* phone developed by Google, that is *5.6"* screen size, *1080x2220* resolution, *440dpi* density. The device is running *Android 13*, also known as *Tiramisu* version, or *API 33*, on *x86_64* architecture. In the really first step while selecting the hardware we would like to use for the virtual device, some device definitions are enabled to run the Play Store software, and so it is for our Pixel 3a device. This is really useful since I had not to download and install manually each applications on the device. Indeed it is made possible using the Google Play store directly from the emulated device.

Once created the new virtual device (in my case called *"Pixel_3a_API_33_-_Data_Leak_Detection"*), it is possible to run it through the Android Device Manager. For simplicity I created a simple *.bat* file to run the virtual device without the need to open Android Studio everytime. Indeed there is a file called *emulator.exe* in a the folder where we installed the Android SDK. We can simply open the emulator by command line passing the name of our AVD and some optional arguments. Here it is how my command looks like:

```
C:\<Android_folder>\Sdk\emulator\emulator -feature -Vulkan -memory 2048 -
    netdelay none -netspeed full -avd Pixel_3a_API_33_-_Data_Leak_Detection
```

Listing 3.1. run_AVD.bat

More specifically the *-feature -Vulkan* argument will force the emulator to do not use the *Vulkan* graphic library. Since Android API 30 lots of android applications, such as Google Chrome, use this library to boost its performance. Anyway in some devices, especially for the emulated ones, might result in a massive slow rendering time. Other arguments are specifying to dedicate 2 Gb of RAM memory for the device, to not impose limits on the network speed and to not simulate any network delay.

Here there is the full property list of the device I used:

Android Virtual Device (AVD) Properties	
avd.ini.displayname	Pixel 3a API 33 - Data Leak Detection
AvdId	Pixel_3a_API_33_-_Data_Leak_Detection
image.androidVersion.api	33
image.sysdir.1	system-images/android-33/google_apis_playstore/x86_64/
tag.id	google_apis_playstore
hw.device.name	pixel_3a
hw.device.manufacturer	Google
hw.cpu.ncore	4
hw.ramSize	1536
vm.heapSize	228
hw.lcd.width	1080
hw.lcd.height	2220
hw.gpu.enabled	yes
hw.gps	yes
hw.accelerometer	yes
runtime.network.speed	full
runtime.network.latency	none

3.1.2 Root privileges on virtual device

After having succesfully created the AVD, it can result useful to get the *root privileges* on the device. In our case it is really mandatory since what we are going to do is intercept the communication protocol in the middle between application and server. As explained in the Section 2.1.3, the client application needs to trust the proxy server, and this is done by checking its certificate. Starting from the release of version *Nougat* (API $\geq$ 24), of Android operative system, it changed the behaviour adopted by android applications in trusting users. If before the application was checking both *user certificates* and *system certificates*, now it will check exclusively the installed system level certificates, therefore root privileges are mandatory.

There are so many tools that let a user obtain the root privileges on his phone. I personally have used the *rootAVD* tool available online [13]. The procedure is really simple, firstly run the command with argument *ListAllAVDs* to list all the AVDs on the machine. We will obtain the full path of our AVDs, that we will pass as argument when executing the tool from command line. The tool will make use the *adb* tool that Android Studio brings together with its development tools, so it is worth it to explain what it is.

3.1.3 Android Debug Bridge (adb)

As I said Android Studio offers a complete set of features to run and test android application. In particular by installing the *Android SDK Platform Tools* package from Android Studio, we will obtain a set of tools that make possible the debug of our application.
One of the most important tools is the Android Debug Bridge, more noticed as *adb*. It is a

command line tool that let the user directly communicate with an Android device, let it be a physical or emulated device. Basically it is a client-server program that runs both on our machine and Android device. There is then a client, that is the command line tool, a daemon *adbd* running in the background of the Android device, and a server managing the connection between client and daemon. For a complete understanding of the adb tool check the Android Developer guide [14].

3.2 Network traffic analysis

Once the virtual device is well configured as described above, let's start introducing some network tool I used for my study cases. Each one of them has some strengths, one tool might works for an application but not work for another. They are listed below in order of complexity. *HttpToolkit* and *BurpSuite* are network traffic interceptors, that means every connection outgoing from our device will be intercepted by these applications, letting the user to inspect more or less accurately how the request were formed. *Wireshark* on the other way is a packet sniffing tool, meaning that it will capture any packet in the network, regardless of the protocol or the port used.

3.2.1 HttpToolkit

HttpToolkit[15] is an open-source tool for debugging, testing and building with HTTP(S). The strength of this tool relies on its simplicity for configuring the environment. Once downloaded the application it requires to be connected to a source of traffic. HttpToolkit provides a variety of available sources, from the most famous browser (Chrome, Firefox, Safari, Edge), up to more complicated environment, such as Docker containers, virtual machines, Android devices or iOS devices. In this case the network source to monitor is the Android virtual device where the applications will run. The program itself will notify the user which are the available sources of traffic at the start up. Since the scope of the study is investigating applications running on the android emulator, it is needed to download from Google Play the HttpToolkit application also on the virtual device. Once installed the android app, the HttpToolkit running on the computer will automatically notice the new source called *"Android device via ADB"*. Just click on it and the whole proxy system is automatically configured.

In this phase it is crucial to understand what is happening:

- HttpToolkit will inject a custom certificate on our device, communicating with it through ADB. Since we also have the root privileges on the android emulator, the certificate willl be placed in the folder of the *System Certificates* that is */system/etc/security/cacerts* in the Android device.

- The respective HttpToolkit Android application will be activated in order to set up a VPN, redirecting the whole network traffic directly via the Android Debug Bridge to

the main HttpToolkit program running on the computer.

- HttpToolkit will copy to the device, again via ADB, a file containing the command used to start up the Android Google Chrome application.

```
chrome --ignore-certificate-errors-spki-list=<hash_digest>
```

This specific command will bypass the Certification Transparency check described in the section 2.1.5. The command looks like the following, where *hash_digest* is the hash digest of the SSL certificate used by the tool.

All these steps are done automatically from the program itself. Indeed this tool requires almost no configuration on the user side, aside from the preparation of the AVD discussed in the above section.

As described in the certificate verification [Section 2.1.2] and MiTM [Section 2.1.3], this is the moment in which the HTTP(S) requests outgoing from the application case are intercepted by HttpToolkit, showing precisely each request detail. Then they are forwarded to the application server. Same happens for the HTTP(S) responses. Moreover in case certificate is rejected by the application, that means some certificate pinning procedure has been adopted, the tool will notify us.

Anyway HttpToolkit is only able to intercept the HTTP(S) protocol, that is enough for most of the Android applications. In case the application will communicate in a different protocol (QUIC for example) the requests are not visible to this tool, so we will need a different network tool.

3.2.2 BurpSuite

Burp Suite is one of the most famous web security suite. It is available in different versions: *Community Edition*, *Professional Edition* and *Enterprise Edition*. The first one is more than enough for personal and research purposes and it is the one I used. It contains the essential toolkit letting the user to be able to set up a proxy and perform a MiTM interception while analyzing the content of the requests passing through the proxy. Differently from HttpToolkit it is more flexible in terms of protocols able to intercept, for example HTTP/3 over QUIC, and it is way more customizable. Indeed it let the user set up specific listeners (address and port), or import/export specific CA certificates with which decode the encrypted messages. Moreover it provide a specific *"Repeater"* tab where the user can manually craft or modify already existing requests and forward them to the endpoint. This last one feature is a routine very useful that I used for every application case.

On the other side, this tool is not auto-configured to intercept traffic coming from mobile device. Everything has to be manually set on both proxy side and mobile device side. Theoretically the configuration steps to enable the interception of HTTPS requests are the same described in the previous section, that HttpToolkit was doing automatically, but instead in this case have to be applied manually [16]:

- From the Proxy options, export the CA certificate in *DER format*. This is the certificate that we will manually have to install as system-level in the Android operative system. Anyway Android will read only a *PEM format* certificate. Once exported the certificate we will need to convert the certificate using the *openssl* tool available for every platform. But this is not enough, in fact the certificate needs to have the filename equal to the *subject_hash_old* value (that is the *hash* of the certificate subject name computed by OpenSSL with the *"old"* version 1.0) appended with *0*. The command lines to achieve the result are:

```
$ openssl x509 -inform DER -in <cacert>.der -out <cacert>.pem
$ openssl x509 -inform PEM -subject_hash_old -in <cacert>.pem | head -1
$ mv cacert.pem <hash_digest>.0
```

  This certificate has to be pushed to the device, and then moved to the */system/etc/security/cacerts/* folder, modifying its permissions. Notice that the ADB tool cannot execute commands as root user while using *play_store* images of Android (kernel images builtin with Google Play application) as in our case. Firstly it has to be pushed on the */sdcard/* location and than moved with the *adb shell -c "mv <source> <dest>"*:

```
$ adb push <hash_digest>.0 /sdcard/
$ adb shell su -c ''mv /sdcard/<hash_digest>.0 /system/etc/security/
    cacerts/''
$ adb shell su -c ''chmod 644 /system/etc/security/cacerts/<hash_digest
    >.0''
```

  Be aware that rebooting the device will remove the certificate and the procedure has to be done again.

- From Proxy options, we have to set up the correct listener. In our case we manually insert the specific IP address of our machine (since the traffic generated from the emulator will have that IP address) or just we can select *All interfaces*. The port can be anyone available, let it be *8082*. Respectively on the Android emulator we have to redirect the outgoing traffic to that proxy. A common proxy setting in the WiFi connection is fine: the IP address will be the one of the computer running Burp Suite, the port will be *8082*.

- Lastly we will have to fix the Chrome behaviour on the application in order to do not apply Certificate Transparency for that specific certificate. In any case this step is optional since we will not be using Chrome, but the Android application itself.

After having configured Burp Suite in this way, by clicking on the *Intercept is off* in the *Proxy* tab, we will be able to intercept any request outgoing from the Android emulated device. As well we can consult the HTTP history of the requests. More than this, we can click *Send to Repeater* from a specific request to manually edit the fields, like Headers or Body, and forward it to the endpoint.

3.2.3 Wireshark

Wireshark is the most famous packet sniffer tool. Differently from the two tools described before, Wireshark is a low level network protocol analyzer meaning that every packet outgoing from the selected Network Card Interface will be captured being able to analyze them. This tool will not differentiate between protocols, every packet will be captured.

Wireshark is a really powerful tool, also able to decrypt TLS communications if provided with the so called *pre-master secret key*. This step is easily possible on Windows, Mac, or Linux operative system by setting an environment variable, so that the browser is enabled to export the secret key used in their encrypted communication. Once exported the pre-master shared keys, it is possible to instruct Wireshark to decode the TLS traffic by using those secret keys. Anyway this is not possible in Android operative system, or at least in this way. In fact it is possible to extract pre-master secret key used by a specific application with dynamic instrumentation tools like Frida or Objection (described in the following Section). See Application Cases (Section **??**for practical examples)

Another possible approach in order inspect decrypted TLS communication in Wireshark is by using some other proxy tool capturing the network traffic at proxy level and capable of exporting the packet logs in *.pcap* or *.pcapng* format. Then is possible to open those logs with Wireshark and analyze every packet captured by the proxy.

In this study Wireshark has been used complementarily to the previous tools HttpToolkit and BurpSuite. In particular since it captures any type of communication protocol, I used Wireshark to know if the Android application was adopting a network protocol different from HTTP(S) or QUIC, and in case I analyze that one.

3.3 Dynamic instrumentation

Instrumentation is a concept of computer programming in which a user obtain informations, or generally trace and profile the behaviour of a software at runtime execution. Specifically **Dynamic instrumentation** tools are able to inject user scripts in the really first phase while running the target application. The injected script is user-defined such that library calls or specific functions can be traced at runtime execution. For this study I used two dynamic instrumentation tools compatible with the Android environment, that are *Frida* and *Objection*.

3.3.1 Frida

Frida is a dynamic code instrumentation toolkit. Frida let the user inject user-defined scripts into native applications running on *Windows*,*MacOS*, *GNU/Linux*, *iOS*, *watchOS*, *tvOS*,*Android*, *FreeBSD*, and *QNX*. This tool is written in *C* language and basically it injects the *QuickJS* JavaScript engine in the target process. This engine will execute the user-defined *JavaScript* script with full access to memory and functions inside the process. One of the strengths of this tool, aside from its extreme compatibility with so many operative

system, is the fact that the *JavaScript* script can directly handle *Java* classes and methods because of the use of Java bridge. So for example in the script we can directly intercept the Java contructs and modify their behaviour.

The installation is pretty simple and can be done via PyPI. The downloaded tool will be the client that we directly use from command line. In fact, since the study deals with Android application, we will need another component of Frida running inside the Android environment, that is *frida-server*. Once downloaded this component will be placed on the Android emulator and get it started. Both *frida-tools* (the client on the host machine) and *frida-server* (the deamon running on the Android device) are able to communicate via ADB. Notice that root permissions are required on the Android device.

There are two ways of injecting scripts in a target Android application:

- The first way is the immediate one. Just write your script in *JavaScript* and let the *frida-tool* inject it into the target application:

  ```
  $ frida -U -l <script.js> -f <andoid.package>
  ```

  The *-U* argument specifies to Frida to interact with the USB device (Android emulator via ADB).
  The *-l <script.js>* argument specifies to load the user-defined *script.js*.
  The *-f <android.package>* argument specifies the target Android application. The application wil be spawned and instantly frozen, the script injection takes place, and the application is then resumed.

- The second way by using the Python API for Frida. The user creates a Python script which will directly handle the Frida injection in its script. Here there is a simple example:

  ```
  import frida, sys

  jscode = ''''''
    Java.perform(() => {

    })
  ''''''

  process = frida.get_usb_device().attach('<android_package>')
  script = process.create_script(jscode)
  print('[*] Running Android Application')
  script.load()
  sys.stdin.read()
  ```

  The Android process is selected by *frida.get_usb_device().attach()* method. The script is a python string containing a *Java.perform(() =>{})* function, and it is loaded with the *process.create_script()* method. The script is then load into the process with the *script.load()* method.

The Frida toolkit comes embedded with some useful tools like *frida-ps* and *frida-trace*, letting the user respectively to check which process is running on the Android device, and to trace library calls while running a specific Android application. For more details check the Frida Documentation[17].
For specific use cases check the Application Case (Section 4).

3.3.2 Objection

Objection is sligthly different tool from the previous one described above. Even if it is powered by Frida, it is defined as *runtime mobile exploration* toolkit. It is specifically developed for mobile applications running either on *Android* or *iOS*. The goal of *objection* is to group together some Frida scripts in order to be accessed by the same tool. Basically it offers nothing more than Frida does, every script can be realized in Frida and be run obtaining the same results, but at the same time it let the user having better times while investigating a mobile application.
It can be install via PyPI and get it started through the command *objection -g <app_package> explore*, where the *-g* argument specifies the application package. The application will be spawned with the tool attatched to it. From this point the user have a set of feature to inspect the application at runtime.

Thanks to the great variety of features implemented in *objection*, this tool let the user to retrieve the same results obtained from static analysis, but at runtime execution. The tool is able in fact to retrieve the full list of classes and methods, and then to hook some of them in order to observe their behaviour during the execution. Some features offered are: dump the whole memory, or just a part of it; search and write specific memory locations; search for specific object instances in the running application. More informations are available on the Objection documentation[18].

For this study purposes I have used *objection* together with *Frida*. Two really useful commands to use are *memory list modules* and *memory list exports <libname>*. The first one will print the list of libraries imported from the application, while the second one will export all the symbols contained in that library. After knowing the name of the library we are sure on which method intercept using either this tool or a Frida script.

3.4 Static analysis

Static analysis is the action of investingating a software without executing the software iself. The work is done on the source code of the application, and in study case on the *Java* code related to the *.apk* of an application.

Starting from any Integrated Development Environment (IDE) the code of the Android application can be written in either *Java* or *Kotlin* languages. This is the source code of the application, that is compiled sequentially in order to obtain the final *.apk* file installable on the mobile device. The steps are:

1. The application source code composed of *.java* or *.kt* files is compiled respectively with *javac* or *kotlinc* compilers obtaining *.class* files containing *Java bytecode*. Multiple *.class* files can be tied up together in *.jar* files. The Java bytecode is executable on Java Virtual Machine (JVM), but not directly on Android devices that uses a different bytecode format, known as *Dalvik bytecode*.

2. The Java bytecode contained in *.class* and *.jar* files is then translated in *.dex* files written in *Dalvik bytecode*, suitable for the Android operative system running the *Dalvik Virtual Machine* (DVM), in the newer versions of Android called *Android RunTime* (ART).

3. At this point the resource files of the application, such as XML, images and layouts are compiled with the Asset Packaging Tool *aapt* tool obtaining a single compiled resource unit.

4. Finally the resource unit and the *.dex* files are put together by the *apkbuilder* tool obtaining the Android Package *.apk* file.

5. At this point the *.apk* is ready to be installed on the mobile device, but not ready to be published on the Google Play Store. In order to distribute the application the developer need to digitally sign it with a certificate. The developer holds the private key, while the certificate with the public key is integrated in the *.apk* file. This process can be done with the *jarsigned* tool.

6. One last step is needed, that is the alignment of the *.apk* file. Indeed Android operative system needs check the authenticity of the application before actually uncompressing the file. For this reason the file is alligned to the byte-boundaries so that the Android OS can directly retrieve the certificate from the *.apk*. This step is made possible through the *zipalign* tool.

Analyzing statically an Android application starting from an *.apk* can be very tedious. All the steps described above have to be reversed in order to obtained the initial source code of the application. Moreover like any other executable format, obfuscation techniques are often adopted to hamper static analysis approach and hide specific routines in the application.

In any case static analysis has been conducted in case some previous tools opened up a path towards a possible data leak discovery.

3.4.1 GDA

GDA (GJoy Dex Analyzer) is a Dalvik bytecode decompiler. It is implemented in *C++* and requires no specific setup and Java VM. The usage is quite immediate, the tool does not require any installation, and it is simply possible to drag an *.apk* file into GDA to start the analysis.
As I said many can be the obfuscation techniques in Android Package files, and most of the times the user have to deal with thousands of classes without any symbols describing classes, functions and methods.
For a specific use case check the Pacer Application Section 4.2.

Chapter 4

Application cases

This chapter deals with every application case discussed in the study. As I explained three different area of interest mobile applications have been investigated: *Weather*, *Health & Fitness* and *Maps & Navigation*. The application case sequence choice has been made by following an increasing complexity order and an increasing amount of informations exchanged.

Generally the applications belonging to the first category do not require any form of user authentications, meaning that few private informations would be shared about the user. For this reason a single application has been taken under examination, that is **iLMeteo**.
The applications that fall in the second category, often have an higher degree of customization in terms of user profile, also implementing some features for user interaction in the application logic. This is the case of **Pacer**, in which the investigation brought some really interesting results.
The last category is potentially the highest source of private user informations retrievable. Maps and Navigator applications require a continuous data exchange between client and server, in oder to deliver specific real time features, like the computation of the fastest path or the current traffic situation delivery. Three different application have been taken in considerations, that are **RadarBot**, **Waze** and **GoogleMaps**.
Obviously the higher the competitor is, the harder the analysis would be. This is the reason why there are so many concepts explained in the Fundamentals chapter (Section 2). Most of them are used not only to improve the application performance, but also to harden the security of the application itself.

The research study conducted application by application is reported in the following sections.

4.1 Application case: iLMeteo

iLMeteo: weather forecasts is one of the most famous weather application available on Google Play Store, counting more than 10 Million downloads. Usually a weather forecast application is one of the really first application installed on a fresh device, if the user

does not want to use the already bundled Android weather application. iLMeteo is a free application and like many other of this kind, a lot of adverticement and monitoring services are implemented together with the weather forecast feature.

4.1.1 Study detail

Once installed the application available from the Google Play Store and opened for the first time, the permission to use geolocalization is asked to the user. An expected behaviour since the application will deal with weather forecasts.

The analysis started with the HttpToolkit software, inspecting every outgoing request generated by the application. Each requests is made using the *HTTPS* protocol, and then totally encrypted, but since we placed our network tool in the middle of the communication we are able to clearly inspect all of them. All the requests observed are grouped in the following paragraphs, basing on the kind of service they are implementing.

Advertising services

Advertising-related requests cover by far the greatest chunk of requests generated, directed towards more than 20 different endpoints. The advertising services provider domains are *criteo.com*, *criteo.net*, *adjust.com*, *taboola.com*, *g.doubleclick.net*, *googlevideo.com*, *googleadservices.com*, *googlesyndacation.com*, and a lot more. Subdomains are not reported, but often a single advertising service provider use different subdomains to deliver the same feature. Requests might assume a certain degree of precision in terms of user information description when communicating with the server. Most of these services will only send generic information on the device used: in the case of *googleads.g.doubleclick.net* the GET request is specifying some information device-related like device model, carrier identifier, Android API level; others are more specific, for example the service offered by *bidder.criteo.com* will include also device ID, screen size, screen orientation, current active session duration and country.

OneSignal: push notification service

A push notification service called *OneSignal* is used in the application, communicating with the endpoint *api.signal.com*. Firstly the application communicates with the server in order to register the current device as a *iLMeteo* application user. Informations sent are *external_user_id* (the id assocated to the user of the application in the context of the *OneSignal* service), *application_id* (the id associated with the *iLMeteo* application), *device_model* (sdk_gphone64_x86_64 for the emulator), *carrier* (*T-Mobile* for the emulator). Once established the link between user and notification service, the subsequent GET requests will retrieve the notifications available for the specific *user_id* sent from the *iLMeteo* service to the *OneSignal* notification service platform.

Firebase: Analytics, Logging and Remote Configuration

The applications makes use of the *Firebase* platform, thanks to which *iLMeteo* implements analytics and logging services. Developed by Google, Firebase keeps track of the user behaviour while using the application. The platform also provides a remote configuration service able to differentiate the layout of the application basing on some informations expressed in the requests. The body of the POST request in this case have a *JSON* structure in which informations on the user device are sent (sdk version, model, hardware, OS build, manufacturer, country, network type).

Android API: Location service

The interaction with Android API happens in order to deliver the geolocalization service, for which the application itself asked the permissions. As HttpToolkit shows, the *gRPC* requests is sent towards the endpoint *voilatile-pa.googleapi/google.internal.android.location.voilatile.v1.VoilaTileService/FindTiles*
The data in this case does not assume any recognizable structure. In fact, the payload of the request seems to be handled with an additional encryption process, over the TLS encrytion.

iLMeteo: weather forecast

Finally the actual weather forecast service behaviour has been investigated. The communication logic is really simple. The client communicates with *iphone.ilmeteo.it/android-app.php* through GET requests. The server implements mainly two methods for delivering weather informations:

1. **getDB**: This method is responsible for filling up the internal database containing the whole list of places for which the user can ask weather informations for. The requests looks like this:

```
GET /android-app.php?method=getDB&table=localita&format=sql&x=<
    getDB_token>&lang=eng&v=4.6&app=com.ilmeteo.android.ilmeteo&force_3h
    =0 HTTP/2
Host: iphone.ilmeteo.it
User-Agent: Dalvik/2.1.0 (Linux; U; Android 13; sdk_gphone64_x86_64
    Build/TPB4.220624.004)
Connection: Keep-Alive
Accept-Encoding: gzip, deflate
```

   The *table* parameter specifies which table has to be retrieved, in this case *localita*.
   The *format* parameter specifies the table format in which the data has to be sent, that is *sql*.
   The *x* parameter identifies an access token for the action *getDB*. If the token is not correct than the request will be rejected by the server.
   The response will contain in its body a list of *SQL* commands, that will fill the application database with the informations relatives to the places available for weather forecasts. The body is essentially a 35 thousands lines of SQL commands: the first line will delete the already existing database, and the second line will create a new database:

```
DROP TABLE IF EXISTS "n_l";
CREATE TABLE "n_l" ("lid" INTEGER PRIMARY KEY NOT NULL DEFAULT (0), "pid
    " CHAR, "rid" CHAR, "nid" CHAR, "nome" VARCHAR, "mare" INTEGER
    DEFAULT (0), "webcam" INTEGER DEFAULT (0), "nome_eng" VARCHAR, "cap"
     VARCHAR, "popolazione" INTEGER DEFAULT (0), "lat" FLOAT DEFAULT (0)
    , "lon" FLOAT DEFAULT (0), "alt" INTEGER DEFAULT (0), "tipo" INTEGER
     DEFAULT (0) );
```

The following lines will be *INSERT INTO* commands to add values to the database. Each line will identify a place available for weather forecast:

```
INSERT INTO "n_l" VALUES(1,'PD','VEN','IT','Abano Terme',0,1,'Abano
    Terme','35031',19726, 45.36, 11.79, 14, -1 );
INSERT INTO "n_l" VALUES(3328,'RM','LAZ','IT','Guidonia Montecelio
    ',0,0,'Guidonia Montecelio','00012',83736, 41.99, 12.72, 105, -1 );
INSERT INTO "n_l" VALUES(279,'LT','LAZ','IT','Aprilia',0,0,'Aprilia
    ','04011',70349, 41.59, 12.65, 80, -1 );
```

The final database will be queried by the mobile application when the user will direcly type in the location he wants to know the forecast about. The database is stored on the devices in the file */data/data/com.ilmeteo.android.ilmeteo/databases/ilmeteo.db*. Each place has an ID location, denoted as *lid*, that will be used to retrieve forecasts for that location. In the listing above, Abano Terme has lid=1, Guidonia Montecelio has lid=3328, Aprilia has lid=279. Notice that the response body does not contains only places near the user location. In the 35 thousands lines there are the cities from the whole world.

2. **situationAndForecast**: This method is responsible for retrieving weather forecasts for a specific location id. The request generated is to get the weather for Rome (id=5913) is:

```
GET /android-app.php?method=situationAndForecast&type=0&id=5913&x=<
    situationAndForecast_token>&lang=eng&v=4.6&app=com.ilmeteo.android.
    ilmeteo&force_3h=0 HTTP/2
Host: iphone.ilmeteo.it
User-Agent: Dalvik/2.1.0 (Linux; U; Android 13; sdk_gphone64_x86_64
    Build/TPB4.220624.004)
Connection: Keep-Alive
Accept-Encoding: gzip, deflate
```

The *type* parameter specifies which type of information the requests is asking for: 0 is for standard locations, 1 is for sea locations, 2 is for oceans, 3 for surfing location.
The *id* parameter specifies the location id.
The *x* is a token that allows the client to use the method *situationAndForecast*. In case this token is not correct, the requests will be rejected by the server.
Other parameters are less importants. The response, in this case, is an *XML* structure containing the information passed to the client in order to visualize the information in the Android application.

4.1.2 Results

The application does not directly handle private informations. Since there is no form of user authentication the only conclusions that can be drawed are traceable back to the device

identifier the various services share.

It is interesting how the weather informations delivery has been implemented. In fact the requests to the *iphone.ilmeteo.it* endpoint specifies the location id of the place the user is looking for the forecast. Since the location database is available and identical for every client, the link between location id and actual location is known. The history of the locations searched by the user might denote the city where the user lives. Anyway the communication protocol is protected by the TLS encryption: every data is encrypted and not visible from the outside. Establishing a Man-In-The-Middle attack the only informations retrievable are those related to the Android device and the locations for which the user have been looking for weather forecasts.

A possible approach to leak private informations would be to link the device identifier to a specific user, but it is something that is not possible within the boundaries of this single application, just because there is no *user* concept in the application logic.

4.2 Application case: Pacer

Pacer: Pedometer & Step Tracker is one of the most downloaded applications among the *Health & Fitness* apps. Developed by *Pacer Health* it counts more then 10 Millions downloads and almost 900 thousands reviews. It delivers pedometer and exercise tracking services. The application let the user create his own profile delivering detailed statistics on his fitness activity. The registration of a new account is mandatory to use the application.

The network communication protocol has been analyzed using both HttpToolkit and BurpSuite. Moreover the tool Frida has been useful for runtime instrumentation, and the GDA tool has been used for static analysis.

The investigation of this application lead in fact, to some important vulnerabilities discovery. All the details are covered in the next section.

4.2.1 Study detail

After having installed the application on the Android emulator, Pacer has been investigated throughout its entire runtime, with the HttpToolkit software as proxy in order to intercept the encrypted traffic.
All the research plan and workflow is described step-by-step in the following paragraphs.

New account setup

At the moment of creating a new account, multiple POST and PUT requests are issued towards the *log.pacer.cc* endpoint. A new session of the user, still unauthenticated, is created. In the first request are sent *app_version* and *device_id* as parameters, and the response will contain the information about the session just created. At this point the user can start the procedure to create a new Pacer account, by typing in the application some information on his person. From the technical point of view two different resources are accessed:

1. */pacer/android/api/v18/accounts/334794565/devices*: This resource contains the informations about the devices used to access that specific account. The body of the request looks like the following:

```
POST /pacer/android/api/v18/accounts/334794565/devices

app_version=p9.10.2
device_id=7021f2bad994fb65
device_model=sdk_gphone64_x86_64
sim_country_code_iso=us
push_service=gcm
platform=android
app_name=cc.pacer.androidapp_play
rom=5.15.41-android13-8-00205-gf1bf82c3dacd-ab8747247%288808248%29
platform_version=13
app_version_code=2022102700
payload=
device_token= <device_token>
has_blood_pressure=false
has_heart_rate=false
has_weight=true
```

2. */pacer/android/api/v18/accounts/334794565*: This resource contains all the informations about the user account. Initially a POST request is issued to create the resource and a subsequent PUT request will add information on the resource. Here is reported the body of the final PUT request.

```
PUT /pacer/android/api/v18/accounts/334794565

avatar_name=sports_running
avatar_path=
display_name=PacerPal
first_install_device_uuid=d9132b20-6a9f-4486-be1f-713a21f113f7
gender=male
language=en
locale=en_US
login_id_alias=p334794565
sim_country_code_iso=us
source=pacer_android
timezone=GMT
timezone_offset_in_minutes=0
unit_type=metric
year_of_birth=1996
```

At this point the user is asked to link his fresh Pacer account to Google and Facebook as identity provider. This will automatically fill in informations like *avatar_path* and *display_name*. If the user agrees, a new PUT request on the same */pacer/android/api/v18/accounts/334794565* resource is issued. The body values are the same, but with additional specifications for *avatar_path* and *display_name*:

```
PUT /pacer/android/api/v18/accounts/334794565

  avatar_name=sports_running
+ avatar_path= image path from <googleusercontent.com>
  content_generated_by=user
+ display_name=Nicola
  first_install_device_uuid=d9132b20-6a9f-4486-be1f-713a21f113f7
  gender=male
  language=en
  locale=en_US
  login_id_alias=p334794565
  sim_country_code_iso=us
  source=pacer_android
  timezone=GMT
  timezone_offset_in_minutes=0
  unit_type=metric
  year_of_birth=1996
```

The account creation and social linking service is done. The Pacer service already got all the information needed to let the user start using the application.

Application startup

The Pacer application is ever running in background, even if the application is being closed or minimized. Everytime the application transitions from the background to the foreground (passing to the top of the screen), a POST request is issued towards the *log.pacer.cc* endpoint. The *app_in_foreground* action is communicated to the server together with *app_version* and *device_id* as parameters, and *account_id* and *install_days* as body fields:

```
POST /pacer/android/api/v18/accounts/334794565/actions/app_in_foreground?
    app_version=p9.10.2&device_id=7021f2bad994fb65

account_id=334794565
action=app_in_foreground
install_days=
```

This request that, up to now, does not provide any useful information. But it is something that we will be using later on in the investigation.

Accessing resources

After having fully configured the account the user can start using the application. Most of the interaction is done towards the server *api.pacer.com* responsible for delivering the services implemented by the application, while few interactions related to logging purposes are directed towards *log.pacer.cc*.
When the user opens the application, he will be showed the homepage containing informations related to his daily activity. At the same time the informations on his account will be retrieved from the Pacer server. Since in the study we are interested in the personal user information, I will explain how these data are obtained.
There are two ways in which the application gathers the information associated to a specific *pacer_id* Pacer account:

1. If the target *pacer_id* is the one associated to the user currently using the application, then a GET request is issued on the resource */pacer/android/api/v18/accounts/<pacer_id>*. This is the request issued when the user opens up the application, visiting the main page of the application. The body of the response is a JSON structure containing the following data (some informations cutted off for length reasons):

```
{
  "id": 334794565,
  "login_id": "p334794565",
  "info": {
    "avatar_name": "sports_running",
    "avatar_path": "https://lh3.googleusercontent.com/a/ALm5wu2j-
    G2Ncb9yct7pyxZr1RuD9UmxVldbgh_BO3MULw=s96-c",
    "display_name": "Nicola",
    "gender": "male",
    "year_of_birth": 1996,
    "source": "pacer_android",
    "email": "alocinalocin@gmail.com",
    "email_status": "active",
    "has_password": false,
    "first_install_device_uuid": "B762315C-EFC0-44BA-9CF8-4CF11534D926"
  },
  "settings": {
    "privacy_type": "private"
  },
  "social": [
    {
      "id": 80332936,
      "social_id": "102620851772027851844",
      "social_id_internal": "102620851772027851844_st_google",
      "head_img_url": "https://cdn.pacer.cc/accounts/334794565/images/
    avatar/2022/11/334794565_af56c5f9-5a23-49f7-ba85-
    f50087bacdff_1668006978150.jpg",
```

```
      "nick_name": "Nicola",
      "social_type": "google"
    },
    {
      "id": 80405740,
      "social_id": "5495598560535308",
      "social_id_internal": "5495598560535308_st_fb",
      "head_img_url": "https://platform-lookaside.fbsbx.com/platform/
    profilepic/?asid=5495598560535308&height=50&width=50&ext=1671029057&
    hash=AeRZcSELt4mZ1V-t9aQ",
      "nick_name": "Nicola Iommazzo",
      "social_type": "fb"
    }
  ],
  "follower_count": "1",
  "following_count": "1"
}
```


These are the informations retrieved by the Pacer server on our specific account, in fact since we are the owner of the account we have the permissions to view some personal informations like the *email* associated to the account and the whole *social* section containing the reference to our Facebook and Google accounts.

2. If an user would like to visit the profile of a different *pacer_id* Pacer user profile the request issued is slightly different. A GET request is made towards the resource */pacer/android/api/v18/accounts/335041780/profile* passing as parameters: *visitor_account_id=334794565&sim_country_code_iso=us*.
   This is the request generated for example when the user wants to visit a profile from his friends list. Since we are visiting an account that we do not own the information retrieved will be less accurated. The JSON response obtained is:


```
{
  "success": true,
  "status": 200,
  "data": {
    "id": 335041780,
    "login_id": "k335041780",
    "settings": {
      "privacy_type": "public",
    },
    "info": {
      "id": 335011000,
      "avatar_name": "icon_dongdong",
      "avatar_path": "https://cdn.pacer.cc/img/avatar/light/04_12.png",
      "display_name": "iommazzo.1693395",
      "gender": "male",
      "year_of_birth": 1980,
      "account_id": 335041780,
      "source": "pacer_android"
    },
    "location": {
      "display_name": "Italy",
      "region_id": "country_it"
    },
    "following_count": "1",
    "follower_count": "1",
    "isPremium": false,
    "is_blocked_by_me": false,
    "social_relationship": "bifollowing",
    "follower_status": "active",
```

```
    "following_status": "active"
  }
}
```

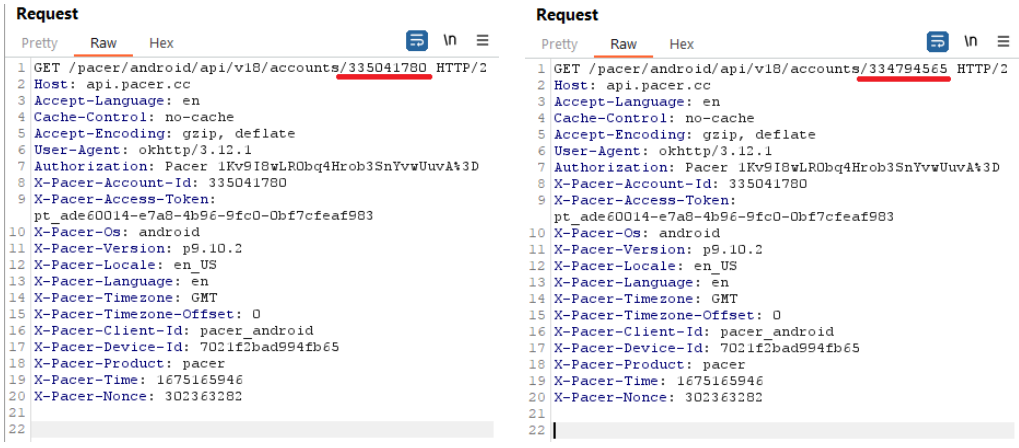

Among the informations retrieved there are not either the email or the social network fields.

So basing on the fact that we are visiting our own account or a different user one, the two requests are generated automatically while using the application. But what is the behaviour of the application if we manually forge a request specifc for the first or second case? For example what happens if we use the first way of accessing resource (aimed to access our own account informations) but using a *pacer_id* different from the one of our account.

Testing unauthorized access #1

For this case I used two different accounts, the first one is linked to the institutional email offered by Sapienza, while the second one is linked with my personal email. The first account with id *335041780* is used to generate the requests; the second account with id *334794565* is the target account. Goal of this test is to understand if it is possible to obtain informations on a different user account, with the same accuracy degree as we were retrieving our account informations. For this kind of tests I used the *Repeater* feature implemented in the BurpSuite software, explained in section 3.2.2.

The first test I tryed was to manually forge a GET request towards the second account resource. The idea is to use the first way of accessing resources but on a different *pacer_id* account. Starting from the standard request generated when visiting our account, I simply changed the path of the resource to */pacer/android/api/v18/accounts/334794565*:



(a) Request to own resource (b) Request to different account resource

Figure 4.1. Test request #1

The result is negative. The request in Figure 4.1(b) does not work, returning in the response the message:
{"success":false,"code":500,"message":"Oops! Could not complete the requested operation,

please try again later and contact support@mypacer.com if the error persists."}.

In any case the behaviour of the application can be understood by carefully reading the error message. It is saying that the request cannot be completed, meaning that maybe it is malformed. There is no message saying that we are unauthorized to access that resource.

Looking at the header fields, there are multiple customized headers of the form *X-Pacer-<value>*. Moreover there is an *Authorization* header that definitely is important. By looking at two generated requests on HttpToolkit to the same resource there are three headers changing every time. **X-Pacer-Time**, **X-Pacer-Nonce** and **Authorization** headers are always different. Definitely those fields have to be coherent each other.

The Authorization header

At this point I decided to explore more in depth how the *Authorization* header is being computed. There are no particular requests the application is performing to retrieve that header everytime, so definitely it has to be computed at runtime by the application.
I started analyzing the *.apk* file of the application in the **GDA** tool explained in Section 3.4.1. With the search string feature provided by the tool, I looked for the string *"Authorization"*, ending up on the following code:

```
private final z b(z p0){
   Integer integer;
   t ot = p0.i();
   String path = ot.F().getPath();   /*path*/
   String str = ot.j();
   String str1 = "";
   String str2 = (str == null)? str1: str;
   String str3 = Pedometer.o(ApiConstant.GROUP_API_TOKEN);
   str = p0.c("X-Pacer-Access-Token");
   String str4 = (str == null)? str1: str;
   str = String.valueOf((System.currentTimeMillis() / 1000));
   str1 = String.valueOf(new Random().nextInt(0x3b9aca00));
   String str5 = this.a(p0);
   String str6 = str;
   try{
      String str7 = str1;
      path = b.a(path, str3, str4, str6, str7, str2, str5);
   }catch(java.lang.Exception e2){
      integer = Integer.valueOf(Log.e("SecurityInterceptorV2", "securityRequest: ", e2));
   }
   path = l.p("Pacer ", integer);
   z$a uoa = p0.h();
   uoa.d("X-Pacer-Time", str);
   uoa.d("X-Pacer-Nonce", str1);
   l.h(path, "hmac");
   uoa.d("Authorization", path);
   uoa = uoa.b();
   l.h(uoa, "requestBuilder\n      .he…on\", hmac\)\n      .build\(\)");
   return uoa;
}
```

Figure 4.2. Private method *b*.

In this code I also found some other strings like *"X-Pacer-Access-Token"*, *"X-Pacer-Time"* and *"X-Pacer-Nonce"*. The semantic of this method is pretty unknown up to now. In the highlighted line we can see that an additional method *a* is called on the object *b* passing as parameter some strings, and obtaining a value called *path*. Looking at the code those strings are:

- *str3* is obtained from the *Pedometer* object;
- *str4* is equal to the *X-Pacer-Access-Token*, if it is not *null*;
- *str6* is equal to *String.valueOf((System.currentTimeMillis() / 1000))*;
- *str7* is equal to *String.valueOf((new Random().nextInt(0x3b9aca00)))*;

- *str2* is obtained from a structure *ot*, if it is not *null*;
- *str5* is obtained from *this* object with the method *a*;
- *path* is obtained from the method *getPath()*.
- The same *path* variable is also the return value of the method in the highlighted line. After in the code this variable is used along with the *"Authorization"* string.

I have also inspected the highlighted method *a*, that is being called passing all the above strings as parameters. This is the relative code:

```
public class b      // class@0015a7
{
|
    public static String a(String p0,String p1,String p2,String p3,String p4,String p5,String p6){
        Mac instance = Mac.getInstance("HmacSHA1");
        instance.init(new SecretKeySpec(p1.getBytes(), "HmacSHA1"));
        instance.update(p3.getBytes());
        instance.update(p4.getBytes());
        if (p2 != null && p2.length() > 0) {
            instance.update(p2.getBytes());
        }
        if (p5 != null && p5.length() > 0) {
            instance.update(p5.getBytes());
        }
        if (p6 != null && p6.length() > 0) {
            instance.update(p6.getBytes());
        }
        if (p0 != null && p0.length() > 0) {
            instance.update(p0.getBytes());
        }
        return (URLEncoder.encode(Base64.encodeToString(instance.doFinal(), 0), "ASCII")).replace("%0A", "");
    }
    public static String b(String p0){
        MessageDigest instance = MessageDigest.getInstance("md5");
        instance.update(p0.getBytes());
        byte[] uobyteArray = instance.digest();
        StringBuilder str = new StringBuilder((uobyteArray.length * 2));
        int len = uobyteArray.length;
        for (int i = 0; i < len; i = i + 1) {
            int i1 = uobyteArray[i] & 0x00ff;
            if (i1 < 16) {
                str = str+"0";
            }
            str = str+Integer.toHexString(i1);
        }
        return str;
    }
}
```

Figure 4.3. Static method *a* and method *b*, of the class *b*.

In this method a new instance of the object *Mac* is retrieved with the method *getInstance("HmacSHA1")*. This object is then initialized starting from the bytes relative to the second string passed as parameter. Then the object is updated with the bytes of other parameters. The return value is a *Base64* encoding of the string obtained by the method *doFinal* called on the Mac *instance*, after having replaced the values *"%0A"* with *""*. This return value seems to assume the form of the value along the string *Pacer*, in the *Authorization* header.
Looking on the Java online documentation the class *Mac* belongs to the library *javax.crypto.Mac* and it provides the functionality for Message Authentication Code. In our case the *"HmacSHA1"* standard algorithm is used.
At this point we are aware of the fact that some custom headers are used to compute the *Authorization* header but we are only sure on few of them. More precisely the *X-Pacer-Access-Token*, the *X-Pacer-Time*, that represents the seconds from the Epoch, and *X-Pacer-Nonce*, that is a random number generated between *0* and *1 bilion* (represented by *0x3b9aca00* in hexadecimal). What it is not known yet are the *p0, p1, p5, p6* values used in Figure 4.3. Going back in the code they belong to specific structures whose values can change at runtime.

Instead of continuing with static analysis approach, I decided to approach the problem in a different way. Since the computation of the *Authorization* header is done request by request, it should be possible to intercept the library calls for *javax.crypto.Mac* and inspect

the parameters passed to the methods directly at runtime. For this reason I used **Frida** for dynamic instrumentation purposes, explained in 3.3.1.

I started creating a script able to intercept the two library calls *javax.crypto.Mac* and *javax.crypto.spec.SecretKeySpec*, overriding the definitions of the methods *getInstance()*, *init()*, *update()* for the *Mac* object, and also overriding the contructor for the object *SecretKeySpec*. All the methods said above are substituded with new methods having the same semantic of the original ones, but just before returning the result they will print the the content of the parameters. Also a support function *bin2String* has been implemented otherwise binary values will be printed instead of strings. The JavaScript code is embedded in a Python script that will handle the Frida process, attatching the script while spawing the process *cc.pacer.androidapp*, as described in Section 3.3.1. The final *jscode* is the following:

```
function bin2String(array) {
  var result = "";
  for (var i = 0; i < array.length; i++) {
    result += String.fromCharCode(array[i]);
  }
  return result;
}

const Mac = Java.use('javax.crypto.Mac');
const SecretKeySpec = Java.use('javax.crypto.spec.SecretKeySpec');

const Mac_getInstance = Mac.getInstance.overload('java.lang.String');
const Mac_init = Mac.init.overload('java.security.Key');
const Mac_update = Mac.update.overload('[B');
const SecretKeySpec_new = SecretKeySpec.$init.overload('[B', 'java.lang.
    String');

Mac_getInstance.implementation = function (str) {
  const result = Mac_getInstance.call(this, str);
  console.log('[+] Entering Mac.getInstance()');
  console.log('[ ] algo: ' + str);
  console.log('[-] Leaving Mac.getInstance()');
  console.log('');
  return result;
};

Mac_init.implementation = function (key) {
  const result = Mac_init.call(this, key);
  console.log('[+] Entering Mac.init()');
  console.log('[ ] key: ' + bin2String(key.getEncoded()));
  console.log('[-] Leaving Mac.init()');
  console.log('');
  return result;
};

Mac_update.implementation = function (bytes) {
  const result = Mac_update.call(this, bytes);
  console.log('[+] Entering Mac.update()');
  console.log('[ ] bytes: ' + bin2String(bytes));
  console.log('[-] Leaving Mac.update()');
  console.log('');
  return result;
}

SecretKeySpec_new.implementation = function (bytes, str) {
  const result = SecretKeySpec_new.call(this, bytes, str);
```

```
  console.log('[+] Entering SecretKeySpec()');
  console.log('[ ] bytes: ' + bin2String(bytes));
  console.log('[ ] algo: ' + str);
  console.log('[-] Leaving SecretKeySpec()');
  console.log('');
  return result;
}
```

Of course we have no control on the runtime execution flow executed by the application, anyway from the previous investigation we know that everytime the application is opened, a POST request is issued to notify the server that Pacer passed to foreground. In the following output log it is possible to show the data associated to that request:

```
[*] Running cc.pacer.androidapp

[+] Entering Mac.getInstance()
[ ] algo: HmacSHA1
[-] Leaving Mac.getInstance()

[+] Entering SecretKeySpec()
[ ] bytes: B7A4DB15-D69A-4C8A-BA68-39E0AA208DB8
[ ] algo: HmacSHA1
[-] Leaving SecretKeySpec()

[+] Entering Mac.init()
[ ] key: B7A4DB15-D69A-4C8A-BA68-39E0AA208DB8
[-] Leaving Mac.init()

[+] Entering Mac.update()
[ ] bytes: 1668595712
[-] Leaving Mac.update()

[+] Entering Mac.update()
[ ] bytes: 336394575
[-] Leaving Mac.update()

[+] Entering Mac.update()
[ ] bytes: pt_11b9df93-167d-4d31-b579-6a0cb2197f10
[-] Leaving Mac.update()

[+] Entering Mac.update()
[ ] bytes: app_version=p9.10.2&device_id=7021f2bad994fb65
[-] Leaving Mac.update()

[+] Entering Mac.update()
[ ] bytes: 84dfbceabc7185c5513137d08a431767
[-] Leaving Mac.update()

[+] Entering Mac.update()
[ ] bytes: /pacer/android/api/v18/accounts/334794565/actions/
    app_in_foreground
[-] Leaving Mac.update()
```

The results obtained are the same obtained from the static analysis approach, but more accurated:

1. The method *getInstance()* is called on the *Mac* object with the string *HmacSHA1* as parameter.

2. A new *SecretKeySpec* is created by passing a string *p1* to it. The *p1* string is exactly *"B7A4DB15-D69A-4C8A-BA68-39E0AA208DB8"*, and in the Figure 4.2 it is identified

by the *str3* variable: the one obtained from the *Pedometer* object.

3. The *Mac* object is then initialized with the method *init()* passing the string of the previous step.

4. The method *update()* on the *Mac* object is called with the string *"1668595712"*. This is the *p3* variable representing the value of the *X-Pacer-Time* header.

5. The method *update()* on the *Mac* object is called with the string *"336394575"*. This is the *p4* variable representing the value of the *X-Pacer-Nonce* header.

6. The method *update()* on the *Mac* object is called with the string *"pt_11b9df93-167d-4d31-b579-6a0cb2197f10"*. This is the *p2* variable representing the value of the *X-Pacer-Access-Token* header.

7. The method *update()* on the *Mac* object is called with the string *"app_version=p9.10.2&device_id=7021f2bad994fb65"*. This is the *p5* variable, relative to the *str2* variable in Figure 4.2. This is the value obtained from the *ot* object.

8. The method *update()* on the *Mac* object is called with the string *"84dfbceabc7185c5513137d08a431767"*. This is the *p6* variable, relative to the *str5* variable in Figure 4.2. This is the value obtained from *this* object, applying the method *a*. This string really seems to be an hash digest of some value. As showed in Figure 4.3 there is another method not yet used. The routine is similar to the one just described, but with just a single string as parameter, and uses the *MD5* algorithm. It is not hard to guess that this is the MD5 hash value of the body of the request. It is easy to double check with any MD5 online hash tool.

9. The method *update()* on the *Mac* object is called with the string *"/pacer/android/api/v18/accounts/334794565/act*
This is the *p0* variable, relative to the *path* parameter in Figure 4.2. This is the value obtained from the *ot* object with the methods *F()* and *getPath()*.

With a dynamic instrumentation approach we got all the values used at runtime by the code showed in Figure 4.2. Anyway we are not sure that the final string computed is actually used as *Authorization* header.

To prove the concept just explained, I created a Java program with the same first method showed in Figure 4.3. The code is reported below:

```
public static String foo(String p0,String p1,String p2,String p3,String p4,
    String p5,String p6) throws NoSuchAlgorithmException, InvalidKeyException
    , UnsupportedEncodingException{

Mac instance = Mac.getInstance("HmacSHA1");
instance.init(new SecretKeySpec(p1.getBytes(), "HmacSHA1")); // PEDOMETER ID
instance.update(p3.getBytes());                               // TIME
instance.update(p4.getBytes());                             // NONCE
if (p2 != null && p2.length() > 0) {                      // ACCESS_TOKEN
  instance.update(p2.getBytes());
}
if (p5 != null && p5.length() > 0) {         // PATH PARAMETERS
  instance.update(p5.getBytes());
```

```
}
if (p6 != null && p6.length() > 0) {      // MD5 BODY (POST, PUT, PATCH)
  instance.update(p6.getBytes());
}
if (p0 != null && p0.length() > 0) {          // URL PATH
  instance.update(p0.getBytes());
}
return (URLEncoder.encode(Base64.encodeToString(instance.doFinal(), 0), "
    ASCII")).replace("%0A", "");
}

public static void main(Strings[] args) {
  String p0, p1, p2, p3, p4, p5, p6;
  p0 = "/pacer/android/api/v18/accounts/334794565/actions/app_in_foreground";
     // p0.i.F.getPath();
  p1 = "B7A4DB15-D69A-4C8A-BA68-39E0AA208DB8";  //Pedometer.o(...)
  p2 = "pt_11b9df93-167d-4d31-b579-6a0cb2197f10";     //"X-Pacer-Access-Token
    ";
  p3 = "1668595712";  //"X-Pacer-Time"
  p4 = "336394575";   //"X-Pacer-Nonce"
  p5 = "app_version=p9.10.2&device_id=7021f2bad994fb65"; // Path parameters;
  p6 = "84df-bceabc7185c5513137d08a431767"; //  md5 body

  String ret = null;
  try {
    ret = foo(p0, p1, p2, p3, p4, p5, p6);
  } catch (NoSuchAlgorithmException e) {
    e.printStackTrace();
  } catch (InvalidKeyException e) {
    e.printStackTrace();
  } catch (UnsupportedEncodingException e) {
    e.printStackTrace();
  }

  System.out.println(ret);
}
```

The *foo* function is copy-pasted from the Figure 4.3, while the values of the parameters are directly typed in the code. Any test can be done with any request showed in HttpToolkit. The fields needed for the computation of the **Authorization header** are **URL Path**, **X-Pacer-Access-Token**, **X-Pacer-Time**, **X-Pacer-Nonce**, **URL Parameters**, **MD5 Body Hash**, and **Pedometer Code** (a value constant for the whole installation of the Pacer application.

Testing unauthorized access #2

Now that we have achieved to forge the Authorization header, we know that in the previous test there was something wrong. Indeed we were modifying the *URL path* (changing the target resource) without changing the value of the *Authorization* header. Using our Java program to forge this header the values to use are:

- p0 = "/pacer/android/api/v18/accounts/334794565", that is the *URL path*.
- p1 = "B7A4DB15-D69A-4C8A-BA68-39E0AA208DB8", that is the *Pedometer code*.
- p2 = "pt_ade60014-e7a8-4b96-9fc0-0bf7cfeaf983", that is the *"X-Pacer-Access-Token"* header.

- p3 = "1675465946", that is the *"X-Pacer-Time"* header.
- p4 = "302363282", that is the *"X-Pacer-Nonce"* header.
- p5 = "". There are no parameters needed for this URL.
- p6 = "". Body not handled by the application in GET requests.

The value returned from our program is *Lgm4YH%2Bthm3zl5NtfCadLgsy6Ho%3D*. This will be the value to use for the *Authorization* header.
Proceeding by forging the request in BurpSuite this is what we obtain:

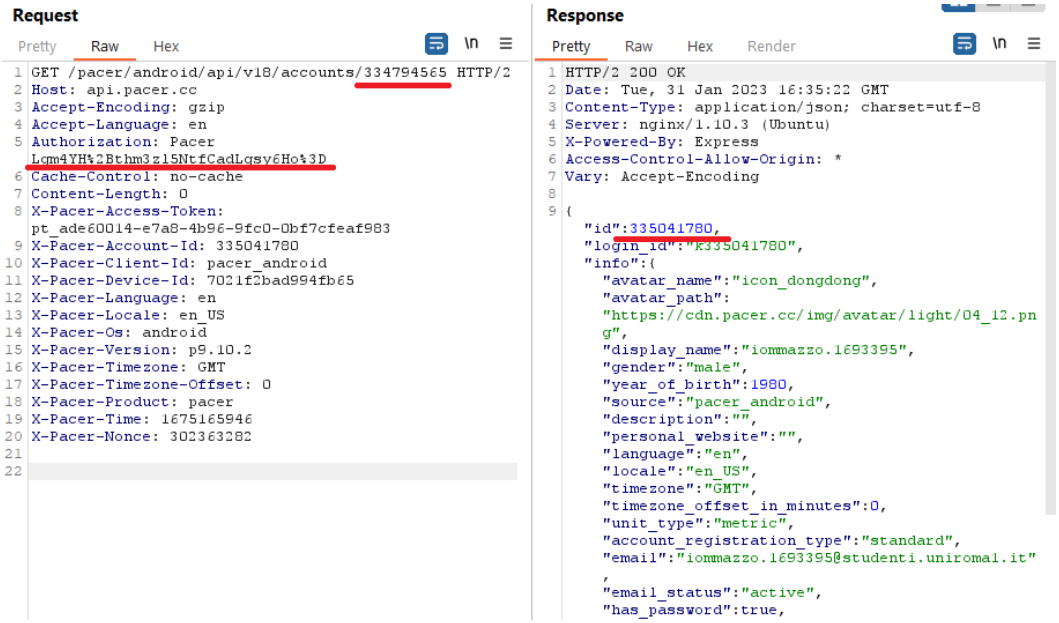


Figure 4.4. Test request #2.

There is no error message showed, meaning that the *Authorization* header we computed it is correct. Anyway the response is not the one we were expecting. Even if the resource on which the GET is performed is the one linked to my personal email (account id *334794565*), the response data retrieved are those relatives to the account we are actually using (account id *335041780*). It is a unexpected behaviour, meaning that the resource specified in the url path is ignored. Indeed a different value is specifying the resource retrieved.

Testing unauthorized access #3

Looking at the forged request, the discriminant value, specifiying on which account we are retrieving informations on, has to be the **X-Pacer-Account-Id**. In the previous request it was still set to the account id linked to the institutional email.
The *Authorization* header is not affected by this change, so we can simply change this value to *334794565*, the *pacer_id* of the account we want to retrieve information on.

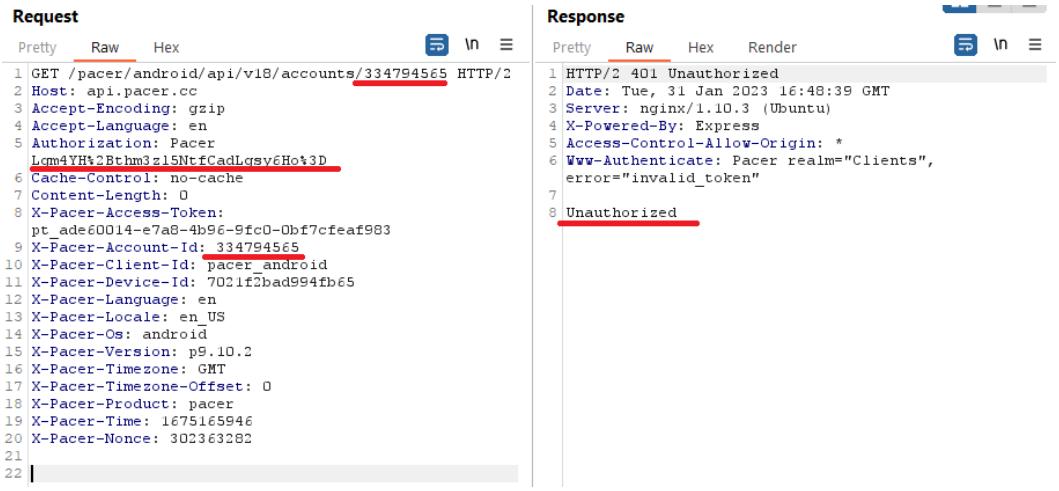
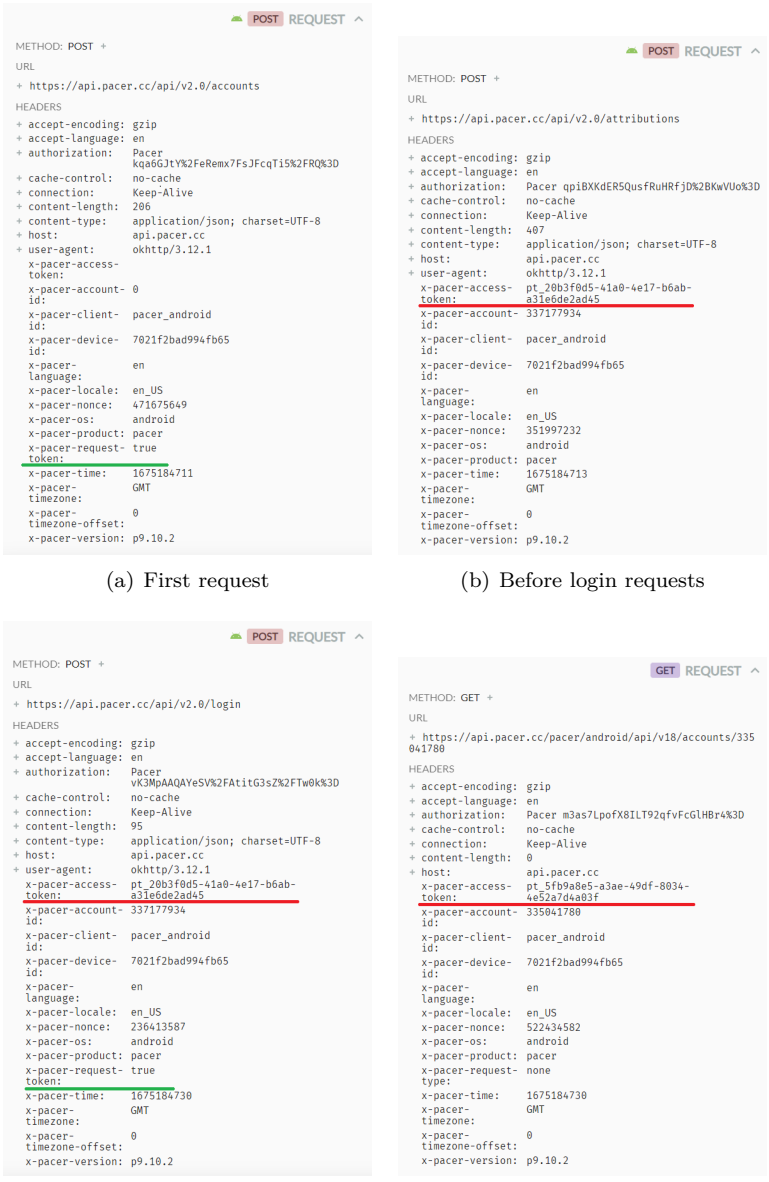


Figure 4.5. Test request #3.

Finally we get an *Unauthorized* message from the server. We have almost checked all the custom headers available. The response *Www-Authenticate* header is saying *error="invalid_token"*. Definitely the *X-Pacer-Access-Token* we are using is not providing us with enough permissions to access informations related to other accounts.

X-Pacer-Access-Token

We already have met this token while forging the *Authorization* header. In any case going back in the communication protocol, I started analyzing all the requests generated in order to understand where this access token is obtained from. This token is assigned at the moment of the user login, after having sent the fields *email* and *password*, or after having completed the OAuth procedure to log in with Facebook or Google. In order to test this behaviour and investigate those requests in HttpToolkit I performed a logout, and again a login on the same account with id *335041780*.
These are four moments in which the investigation has been performed: during the first request generated without any access token, before the login, during the login action, and after the login:



(a) First request (b) Before login requests

(c) Login request (d) After login requests

Figure 4.6. X-Pacer-Access-Token investigation

During the first request [Figure (a)] the POST request has no access token. In this moment the client is actually asking the server for a new one, in fact the header **X-Pacer-Request-Token** is set to *true*. The response to this request contains a fresh access token, to use in the pre-login phase.

Before the user actually logs in [Figure (b)] the *X-Pacer-Access-Token* used is the one retrieved from the previous step.

While performing the *login* [Figure (c)] a POST request is issued communicating *email* and an hash digest of the *password*. The access token used is still the one received at the previous step. Anyway after the login we will need a new access token, associated instead with the account just logged in. Even in this case then the **X-Pacer-Request-Token** is set to *true*.

The response to this login request contains the actual access token. After having performed the login action [Figure (d)] the *X-Pacer-Access-Token* assume the value obtained from the login response. From this moment going on, the value of the access token will be this one for every request.
Investingating the different log in phases a new header is discovered: the **X-Pacer-Request-Token**. Clearly it is used when the client is requesting a new access token, exactly in the Figures (a) and (c).

Final unauthorized access

Now that I have known about this new header **X-Pacer-Request-Token**, I have tryed to forge a request, still having the *X-Pacer-Access-Token* set as standard, but adding a new header that is the *X-Pacer-Request-Token*. If by any chance this last header is being evaluated without actually checking the *X-Pacer-Access-Token* we might bypass any authorization mechanism tied to the access-token. This is the request forged:

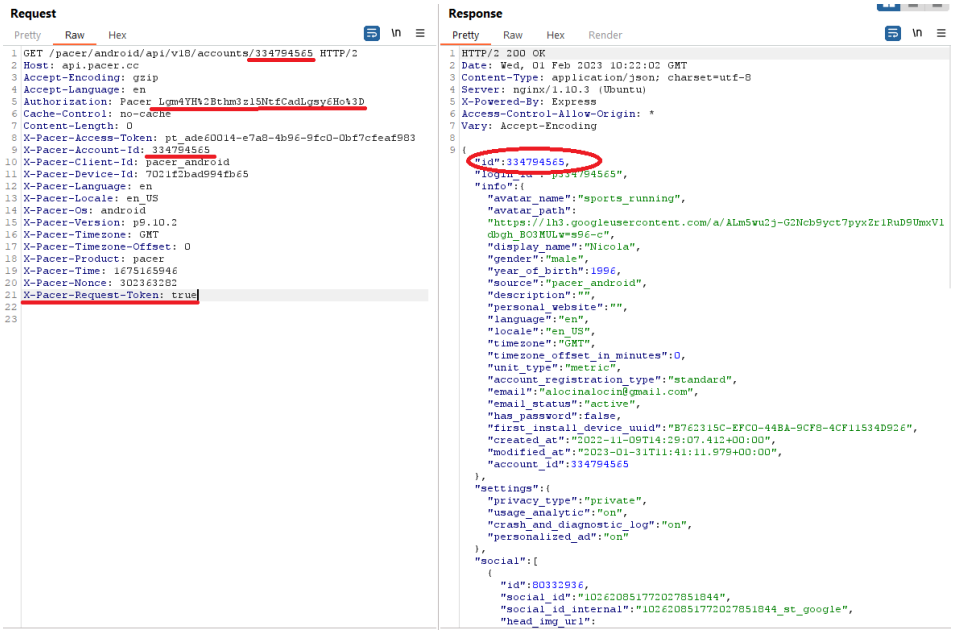


Figure 4.7. Test request #4.

Here we go. The response showed on the right side of the picture contains information about my personal account (id *334794565*) comprensihive of email, gender, picture, social media identifiers. Notice that this is not the case in which I am visiting another user profile, that is a legit action to perform in Pacer: I could have simply visit the account id *334794565*, but the informations retrieved do not contains any personal informations, as showed in the Accessing Resource paragraph of this Section. With this procedure we are asking to the server informations on an account, that we do not actually own, as owner of that account. It is clearly something that should not happen.
Moreover combining this last **X-Pacer-Request-Token**, with the forge of the **Authorization** header described above, we are literally able to control another account as we were its

owner and without needing the access credentials.

Changing setting of another user account

In this paragraph we will push the investigation beyond the scope of accessing the resources. I already have shown how informations can be retrieved on an account that we do not own. Since the *X-Pacer-Request-Type* let us to access information otherwise not accessible, and the *Authorization* header let us to visit every path explorable in the server, why not to try if we can actually forge some POST request, for example modifying to settings relative an another account.

The setting chosen to work on is the Privacy value of an account. On most social platforms an user can decide if its account is *private*, meaning that the informations tied to his account are visible only to his friends, or *public*, meaning that every user can view its informations. Thanks to HttpToolkit, and simulating this setting changing on the emulator, we identified the POST request to be directed towards the resource */api/v2.0/accounts/<pacer_id>/settings/<section>*, where *<pacer_id>* is the target id account (in this case our target account is *334794565*), and *<section>* can assume one of this values [*gps*, *privacy*, *workout*], defining the section of the settings we are going to modify. The url path is given. The forged request will have the following headers:

1. *X-Pacer-Access-Token* is the one of our account.
2. *X-Pacer-Request-Token* set to *true*, this will bypass the check on the access token.
3. *X-Pacer-Nonce*: we can use the older values for the nonce.
4. *X-Pacer-Time*: we can use the older values for the time.
5. *X-Pacer-Account-Id*: we can leave this value to the target id account, and see how it works.
6. Every other *X-Pacer* header can be left as it was.
7. Since this is a POST request there will be also a body. In this case the body of the request will contain the whole list of settings belonging to the section Privacy. From HttpToolkit we know that the body is a JSON structure of the form:

   ```
   {"crash_and_diagnostic_log":null,"personalized_ad":null,"privacy_type":"
       public","usage_analytic":null}
   ```

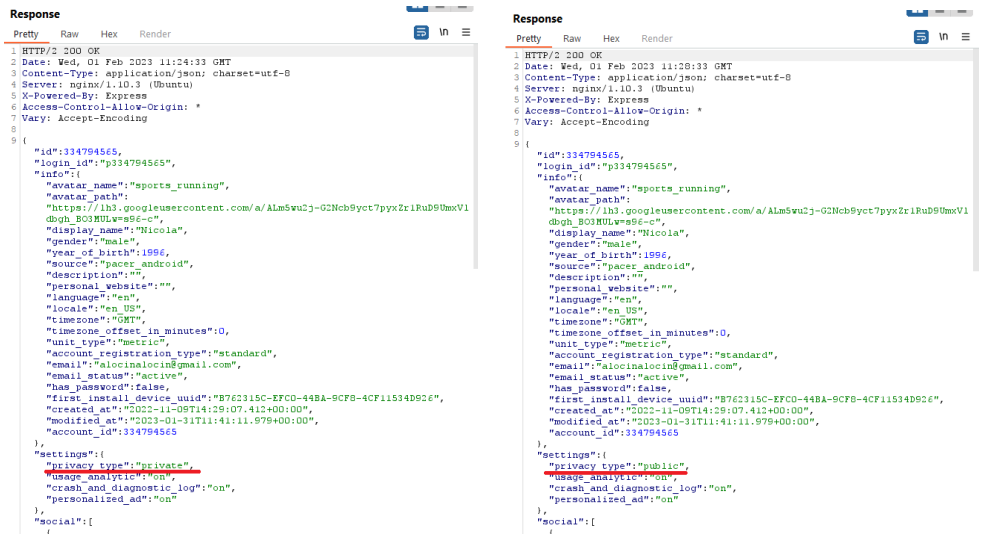
   where the specific setting values we are going to change have to be set, while every other value is not changed is *null*. Since on our target account the privacy setting is set to *"private"*, try instead to change it in *"public"*.
8. Finally the *Authorization* header has to be forged with our Java program, keeping in considerations the values of *url path*, *Pedometer code* (still the same leaked with Frida), *X-Pacer-Access-Token*, *X-Pacer-Time*, *X-Pacer-Nonce*, *path parameters* (there are no parameters for this requests), *MD5 of body request*. We already got all of these value.

We only miss the *MD5* hash of the body, easily computable with any MD5 online tool. In this case the hash digest of the body is *34303e67eaa68ec11ab5c6556c74f9e4*. The resulting *Authorization* header is *lXT8UuMb%2F49KgK8ZRCAloUoPbV4%3D*.

The request and response generated with in BurpSuite are:

Figure 4.8. Privacy setting change request.

The response expresses that the request was successful.
By comparing the informations retrieved on the target account before and after our forged POST, we can see the differences:



(a) Before the POST request (b) After the POST request

Figure 4.9. Before and after the privacy settings change

The procedure described was actually able to change the privacy settings of a target account, by only knowing its *pacer_id*.

The procedure is therefore valid for every other kind of settings change.

Training paths

Keep going with the private informations investigation, another feature of the Pacer application has been analyzed, that is the *Training Paths* sharing. Each user can record its training

session and save the log of the session, comprehensive of calories, distance, and time the session lasted. The application handle two types of training paths:

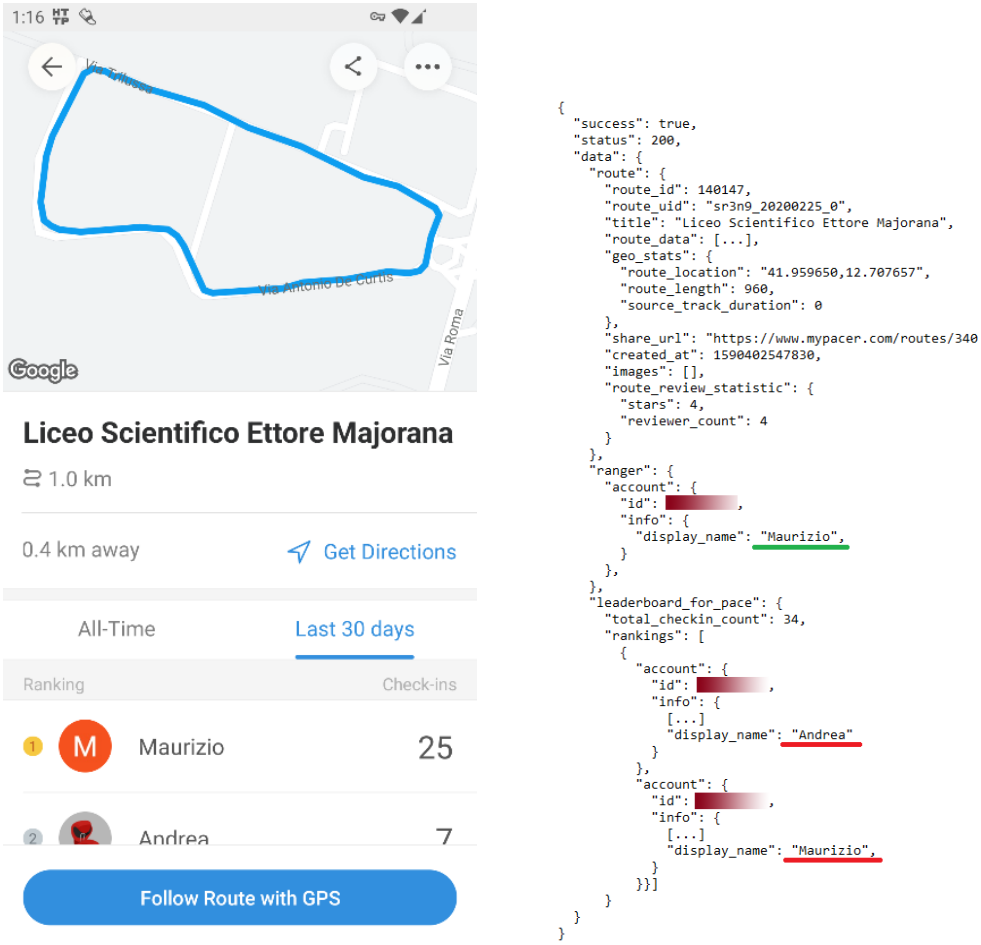
1. *Locally stored training paths*: Every training routine manually started by the user will be saved offline on the device, and it is possible access it only from the device that logged that training session.

2. *Public shared training paths*: The locally stored sessions can be chosen to be publicly shared. These ones are accessible from the *Feed* tab by any user following our account, from the *Explore* tab by a user that is near the location of the training path, or by visiting the user profile.

The first type of training paths cannot be inspected without a physical access to the device. On the other way the second ones are accessible, in fact the user that published them have decided to share it with the community. In this sense there is no actual leak of private informations obtainable from these, since the user has spontaneously chosen to share them.

Obtaining user account IDs

Up to now we only miss a single component in order to retrieve personal data on a target user, that is its *account_id*. As introducted before, Pacer in the *Explore* tab of the application, provide a map where it is possible to visualize the training paths published by the users. Tapping on one of these paths the informations on the path and on the users that used that paths as training route are shown. In the Pacer application are not directly visible the users *account_id*, but on the HTTPS response they are. The following picture shows both the Pacer application view and the relative body HTTPS response:



(a) Pacer application response view (b) HTTPS response

Figure 4.10. Visiting a path from the Explore tab of the application

As the picture (a) describes, from the Pacer application can only be seen which are the top ranked users. Even by tapping on a profile the relative *account_id* is not shown. On the right side picture (b) it is indeed showed the JSON code in the body of the HTTPS response, in a redacted form and obscurated from the *id_account* of the relative users. The first one is the creator of the route, while the below ones are from the ranking list.

This is only a possible way of retrieving *account_id* values of the Pacer user. In fact every comment and post from the *Explore* and *Feed* tabs in the Pacer application will deliver us the *account_id* of the user. Once obtained the user account id, the whole procedure for accessing other users resources described above can be achieved.

4.2.2 Results

Here are reported the results obtained by the Pacer application study case.

Private informations leaked

The Pacer application is exposed to users private informations leak. Given a Pacer user *account_id* it is possible to obtain the following private informations:

- Profile picture.
- Gender.
- Email address.
- Device UUID.
- Country and Timezone.
- Facebook and Google social data linked including:
  - Social nickname.
  - Social profile picture.
  - Social ID used for the Pacer app.

Other vulnerabilities

The study conducted shows the presence of an important vulnerability that allows an user to bypass the access-token based authorization mechanism, and access the private resources of any Pacer *account_id* user.
This vulnerability exploitation might bring to the loss of control for any user on its own account. Any account can be accessed by simply having the relative *account_id*.
The source of the vulnerability relies in how the back-end system handle the header *X-Pacer-Request-Token*.

4.3 Application case: RadarBot

RadarBot: Speed Camera Detector is an Android application able to retrieve data about traffic and speed cameras while driving. It counts more than 50 millions downloads on Google Play and it is the first application analyzed of the *Navigation & Maps* category in this study. This application will notify the user about every alert regarding the driving and for this reason it requires a constant interaction between mobile app and the server delivering the service.

4.3.1 Study detail

Once installed RadarBot and started the application for the first time, the user is asked to register a new account though the Google or Facebook identity providers. In this application case the user cannot register a new account without having completed this step, so I proceeded with linking my personal Google account. At this point the account is created, and the application is ready to use. The application starts with the visualization of our current position on a map.

The first tool used to inspect the application behaviour is *HttpToolkit*. The application interacts with different enpoints using the HTTPS protocol, that means every request is encrypted, but since we placed our proxy tool in the middle of the communication we are still able to investigate the application behaviour. Most of the requests are directed towards two endpoints:

1. **clients4.google.com**: This is the endpoint relative to the GoogleMaps API used in this application to retrieve the portion of map to display in the application. Since the user will constantly change position while driving, lots of this kind of request will be issued to retrieve the map. In any case the data contained in the response is only related to the representation of the map itself (streets and buildings).

2. **radarbotservices.com**: This is the real server delivering the RadarBot services. Multiple POST requests are done on different resources.

Among the other requests we can notice the *OneSignal* notification service and the *Firebase* logging service (same for *iLMeteo* application).

RadarBotServices requests

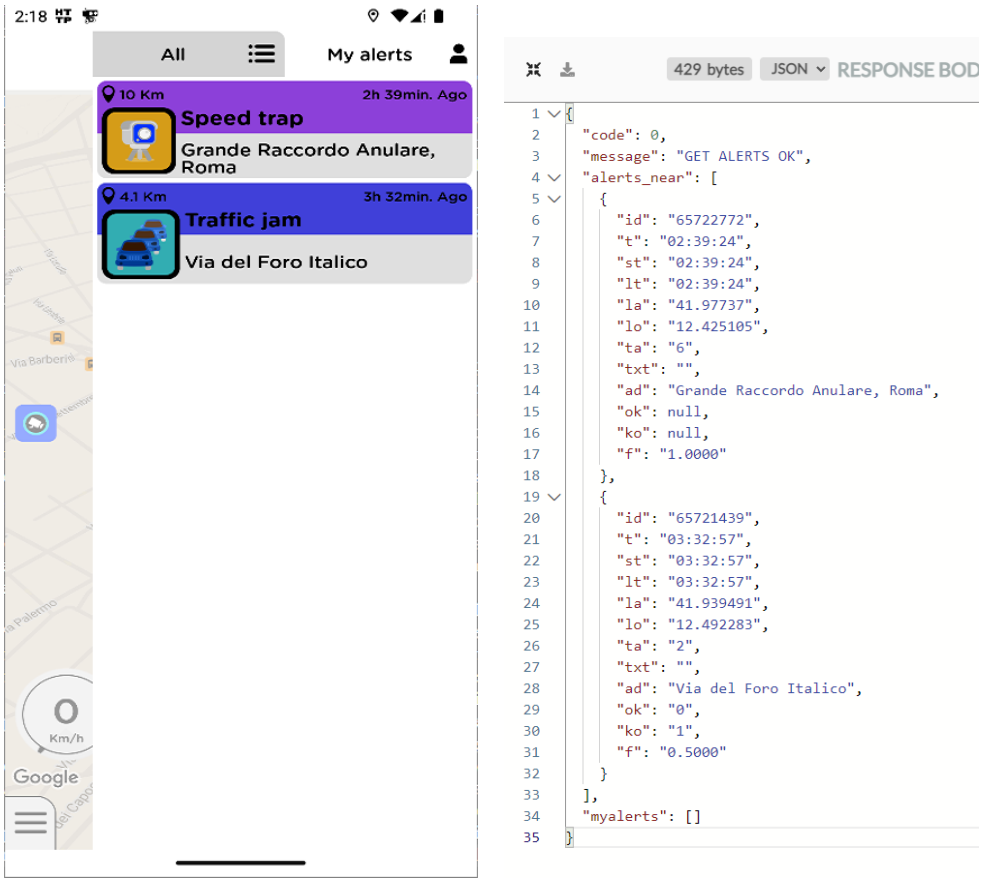
radarbotservices.com is the endpoint responsible for delivering most of the feature implemented in the application, for example the alerts delivery system or the possibility to create a new alert on the map.

The first request issued while using the application is a POST on the resource called **ws_check_update_database.php**. In the body of the request we can notice the *device_id* and the *firebase_id*. Probably this is the request responsible of notifying the server that the device is using the RadarBot service, and at the same time the version of the software is checked. As response we got a simple *"code": 0,"message": "WS OK","new_version": 1* message.

Right after a new POST request is issued on the resource **ws_get_licence.php**. As before the body request contains *device_id* and *firebase_id*. In the response this time we get some identifiers relatives to the licences used by the application. In fact RadarBot offers either a free or a premium service. With this request the application is checking if a premium licence is active on our device.

At this point the application will update the information on the user using RadarBot. Therefore another POST request is issued towards **ws_user.php** sending again in the body again informations on *device_id* and *firebase_id*, but also private informations such as *latitude*, *longitude* and *email*. The response message received is *"code": 0,"message": "User updated OK"*.

The above described requests are done in the initialization phase of the application. After this setup, the application behaviour enters a loop where two requests are repeatedly issued. The first request is a POST on **ws_get_country.php** where the application notify the server of some movement, if happened. In the body there are both *latitude* and *latitude* coordinates. The response will contain the country where we currently are. The message is *"code": 0,"country_code": "it"*.
The second is request is another POST on **ws_get_alerts.php** where the informations regarding speed cameras and alerts are retrieved by the client. Even in this case the *latitude* and *longitude* coordinates are sent to the server. The response is depending on the alerts database. In the pictures above there is an example:



(a) Radarbot application response view (b) HTTPS response

Figure 4.11. Retrieving alerts in RadarBot application

As you can see in this example there are two items in the *alerts_near* list. The first one is relative to the alert on *Grande Raccordo Anulare*, while the second one is relative to the alert in *Via del Foro Italico*. Informations on the alerts are comprehensive of *alert_id*, *latitude*, *longitude* and *time* since report.

Since we are interested in user informations, the application investigation continues looking for any evidence of user ids. After having checked some other less relevant requests (like the ones generated when rating alerts on the map, creating an alert or removing an alert), we notice that there are no informations on any other user of the Radarbot service. The only information exchange at which the client takes part are the one related to itself.

4.3.2 Results

Sensitive data

In terms of sensitive and private data, the informations involved in the communication protocol are:

- **device id**: It is present in the body field of each requets generated.

- **firebase id**: It is used as actual user id for every request. At this id are associated also information on the device (hardware, device model, os build)
- **latitude and longitude**: As described in the previous section, these fields are sent in the requests *ws_get_country* and *ws_get_alerts*.
- **email**: The user email is sent in the request *ws_user* in the initialization phase of the application.

All these information are related to the user currently using the RadarBot application. In any case the communication flow is encrypted by the use of the HTTPS protocol, so every discussed information is obscured to third parties. The presence of a Man-In-The-Middle, and therefore of a proxy installed on the device redirecting the traffic to a controlled server, is still able to retrieve the informations above.

No private informations are available on the other users of the Radarbot service, since they are not involved in the communication protocol.

Other vulnerabilities

While looking for any information user related in the network communication flow, some specific behaviour of the application have been investigated. For example the possibility to remove alerts from the map created by some other user.

391 bytes URL-Encoded REQUEST BODY

ws_version	2
rb_version	189
password	radar_asm256K!@
pais	it
id_dispositivo	8F66-351D-877E-5ADD
p	C7zfqMfdsLySfwltBNu6uNQHdIvkvdvIhq/dGwyp2mj/4trauNk=
os	0
up	0
id_firebase	dEqM7sgPTky15kv50VM9Zf:APA91bEgagjzqwUUBWhJUaxmP'
latitud	41.902783
longitud	12.496365
ts	0

Figure 4.12. Radarbot body request fields.

From this point of view the application relies on two variables that are present in the body of each request generated by the client, showed in the picture above. The first one is *id_firebase* that is evaluated as actual user id. The second one is the *p* field containing a Base64 value sent in each request.
In the case in which the **id_firebase** field is not sent in the body, the request is ignored, obtaining a void response from the server.
On the other side, if the **p** field is not sent in the body, the related response will contain the message *"code": 2,"message": "BBDD PARAMS WS"*.

In order to test any other vulnerability affecting the application is needed to forge a request that will be accepted by the server, and then we need to forge a new value for the field *p*.

Forging requests

In order to forge any valid request, the field **p** has to be set accordingly. Starting analyzing two identic requests on the same path (same headers, same body values), the field *p* assumes two different values.

To understand how this field is computed at runtime I gave a look at the source code of the application using the tool *GDA*. By a searching the string *"id_dispositivo"* I rapidly came up into the piece of code where the body fields are taken into account. This is the code:

```
public static p7$c n(){
   byte[] uobyteArray;
   int i6;
   int i7;
   p7$c uoc = new p7$c(null);
   int i = 2;
   uoc.a("ws_version", String.valueOf(i));
   uoc.a("rb_version", String.valueOf(181));
   uoc.a("password", "radar_asm256K!@");
   String g = i6.g;
   if (g != null) {
      uoc.a("pais", g);
   }
   uoc.a("id_dispositivo", d.b(ITApplication.getContext()));
   g = RadarApp.d().getApplicationContext().getPackageName();
   int i1 = 0;
   try{
      uobyteArray = new byte[i1];
      uobyteArray = g.getBytes("UTF-8");
   }catch(java.io.UnsupportedEncodingException e2){
      e2.printStackTrace();
   }
   Random random = new Random();
   int i2 = Math.min((random.nextInt(8) + 8), uobyteArray.length);
   byte[] uobyteArray1 = new byte[i2];
   random.nextBytes(uobyteArray1);
   for (int i3 = i1; i3 < uobyteArray.length; i3 = i3 + 1) {
      int i4 = i3 % i2;
      i4 = uobyteArray1[i4] ^ uobyteArray[i3];
      uobyteArray[i3] = (byte)i4;
   }
   byte[] uobyteArray2 = new byte[((i2 + 1) + uobyteArray.length)];
   uobyteArray2[i1] = (byte)i2;
   int i5 = i1;
   i6 = 1;
   while (i5 < uobyteArray.length) {
      if (i5 < i2) {
         i7 = i6 + 1;
         uobyteArray2[i6] = uobyteArray1[i5];
         i6 = i7;
      }
      i7 = i6 + 1;
      uobyteArray2[i6] = uobyteArray[i5];
      i5 = i5 + 1;
      i6 = i7;
   }
   uoc.a("p", Base64.encodeToString(uobyteArray2, i));
   uoc.a("os", String.valueOf(i1));
   m om = m.n();
   while (i >= 0) {
      Objects.requireNonNull(om);
      if (!q.q[i]) {
         i = i - 1;
      }else {
         break ;
      }
   }
   uoc.a("up", String.valueOf((i + 1)));
   String str = d.a();
   if (str != null) {
      uoc.a("id_firebase", str);
   }
   return uoc;
}
```

Figure 4.13. Radarbot source code method.

The highlithed line shows the string searched. As we can see there are all the main fields of the body: *ws_version*, *rb_version*, *password*, *pais*, *id_dispositivo*, *p*, *os*, *up*. Specifically the field *p* seems to be computed starting from the bytes of the string relative to the RadarBot package name. Random values are then computed and integrated in some way in the byte array. At the end the value obtained is encoded in Base64 and assigned at the field *p*.

At this point we know exactly how the *p* value has been computed. In order to generate more valid values I created a Java program with the same line codes relative to the computation of this value. The code is omitted since it is just a copy-paste from the source code with some adjustment. In this way we are able to compute an infinite amount of values for the variable *p*.
Notice that the only values that contributes to the final value of *p* are: the package name (*"com.vialsoft.radarbot_free"*) and some integer values fixed or computed at runtime from random values.

By changing the field *p* of some already working requests with the value generated by our Java program, the requests are being accepted, meaning that our program is working and generating good values for the field *p*. In particular there is no time-relative constraint

on this value, meaning that once obtained one it can be reused an infinite amount of time.

Regarding to the **firebase_id** value there is nothing we can do. It is an user id assigned by the Firebase service, and the RadarBot server uses it as user id.

Test: remove other users alerts

Now that we are able to manually forge requests, the investigation continues on testing some edge cases. For example the case in which an user request to remove an alert created by another user. We already have seen in Picture 4.11, how an alert identifier can be retrieved while using the application.
When the user asks to remove an alert that he himself created, a new POST request is issued on the path **ws_remove_alert.php**. Another value has to be added in the body that is *id_alerta*, representing the id of the alert that the user wants to delete. This is the forged request and the relative response, obtained using the *Repeater* tab from the *BurpSuite* tool:

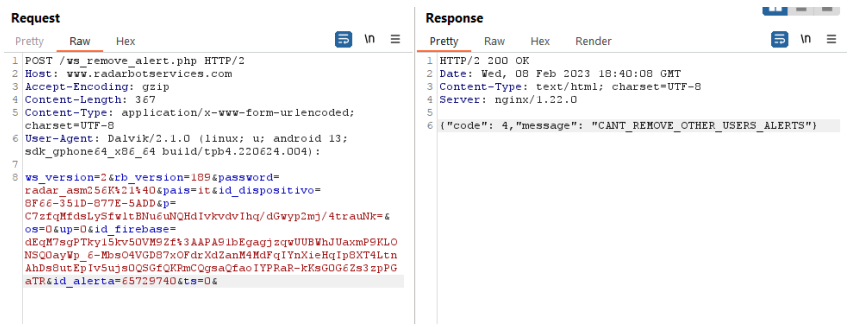


Figure 4.14. Forged request and response to remove alerts.

As you can see the request is accepted by the server, but the response shows the message *"code": 4,"message": "CANT_REMOVE_OTHER_USERS_ALERTS"*. So the server is recognizing that the alert has not been created by the user that requested the action, and it is blocking the action.

Test: modifying user informations

Another vulnerability check has been conducted on the request **ws_user** where the client communicates the update relative to a user profile. For example a possible request might be to modify the email relative to a user profile. In the picture below I tryed to change my personal email into the institutional email.

Anyway in this case, even if the request is being accepted by the server showing the message *"code": 0,"message": "User updated OK"*, there is no actual change in the server database or in the client application. In fact the email it is not changed, and the future request will still contain my personal email data.

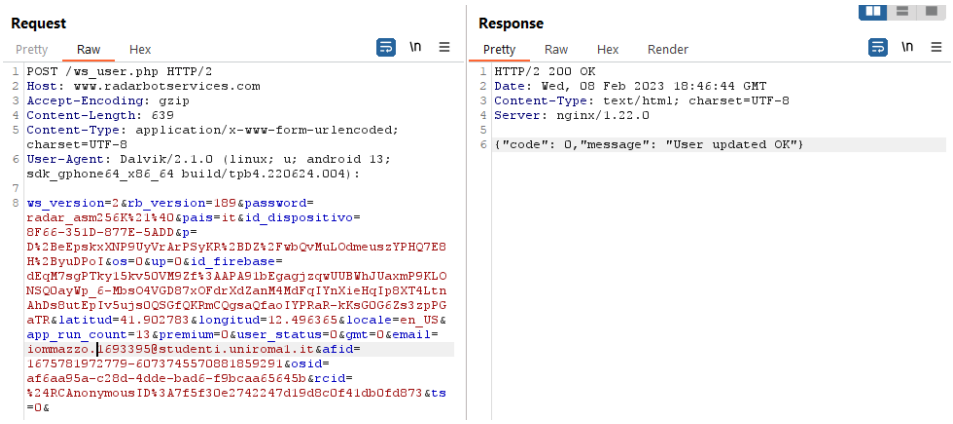


Figure 4.15. Forged remove alert request and response.

4.4 Application case: Waze

Waze GPS & Live Traffic is the second Android application analyzed of the category *Maps & Navigation* in this study case. It is the most famous application delivering real traffic information. downloaded more than 100 million times. Among the main features of the application provided by Waze, there are the real time traffic congestion information delivery and the in-app instant messaging between Waze users. While using this application users can in fact discover new events happened in real time, and directly contact other users to retrieve information on the traffic. This last feature is enabled only to a experienced Waze users that have gained enough points using the application, but the information on the traffic are retrivable in any case.

4.4.1 Study detail

After having downloaded the application from the Google Play store and installed it on the device we are ready for the investigation on the network communication generated by Waze. As first step the *HttpToolkit* tool is started in order to intercept the outgoing network traffic from the Android emulator running the application.

Once started Waze, the application notifies the user the needs of the geolocalization permission. After having accepted these permissions, Waze will start downloading the resources needed for the correct visualization of the application user interface: logos, images, and sounds pack are retrieved by contacting the server *ads-resources.waze.com*. From the technical point of view, the process is done through multiple GET requests on every resource needed.

Once the whole resource pack is downlaoded, Waze will ask for the registation of a new account, to log in if the user is already registered. or using the user's Google account. For this specific case we just create a new account using my institutional email offered by Sapienza. The application will send an email to the relative address, and the user will have to type in the code received in the email. At this point the new account is created and the

user can start to use the application.

The process seems to be standard for an Android application. By looking at the requests generated in *HttpToolkit* we immediately notice that the communication protocol is a bit different from the previous analyzed applications. In fact the application communicates with just one endpoint using the HTTPS protocol, that is *rtproxy-row.waze.com*. Even if the application is not being interacted by the user, continuous requests are generated in order to track the user movements. Waze really behaves as a real-time application transimitting a constant amount of data to the relative server.
The requests generated are all POST requests, directed towards only three resources, that are */rtserver/distrib/static*, */rtserver/distrib/login* and */rtserver/distrim/command*. By looking at the way in which these requests are formed we can notice some of the standard HTTPS headers together with some custom additional header. The custom header *x-waze-network-version* is probably indicating the TLS version used (1.3) while the header *x-waze-wait-timeout* is probably indicating the maximum time to wait for the response. Nothing fancy for the headers.
On the other way the requests body is more complex. The whole informations transmitted are contained in the body of the POST requests through a particular encoding called **ProtoBuffer** (explained in Section 2.2). The body structure anyway, seems to be depending on which one of the three resources is being contacted. For this reason I decided to split the investigation on the requests in three different logical phases of the runtime execution:

1. **Pre-Login phase**: This is the initial phase in which the user is not yet logged in. The application is contacting the */rtserver/distrib/static* resource.

2. **Login phase**: This phase represents the process the application performs to retrieve the user session data. This has not to be confused with the action of typing in the access credentials. The login phase is executed every time the application is quitted and reopened. The user opening up the application is not asked to type in credentials, but the login phase is done automatically. The only way to let the application forget the user credential is by manually performing the logout action in the application. In this case the application is contacting only the */rtserver/distrib/login* resource.

3. **Post-login phase**: This is the phase in which the user is logged in and the application is running either in background or foreground. From this moment going on, the */rtserver/distrib/command* resource is the only one the application will communicate with.

As you can see the three phases are rapidly performed at the start up of the application and captured from HttpToolkit:

Method	Status	Source	Host	Path and query
POST	200		rtproxy-row....	/rtserver/distrib/static
POST	200	?	rtproxy-row....	/rtserver/distrib/login
POST	200	?	rtproxy-row....	/rtserver/distrib/command
POST	200	?	rtproxy-row-l...	/rtserver/distrib/command
POST	200	?	rtproxy-row....	/rtserver/distrib/command
POST	200	?	rtproxy-row....	/rtserver/distrib/command
POST	200	?	rtproxy-row....	/rtserver/distrib/command
POST	200	?	rtproxy-row....	/rtserver/distrib/command
POST	200	?	rtproxy-row....	/rtserver/distrib/command

Figure 4.16. Requests captured in HttpToolkit.

Pre-Login phase

Just opened Waze, the application is in the pre-login phase. A single POST request is issued on the resource */rtserver/distrib/static*.

The body request looks like the following:

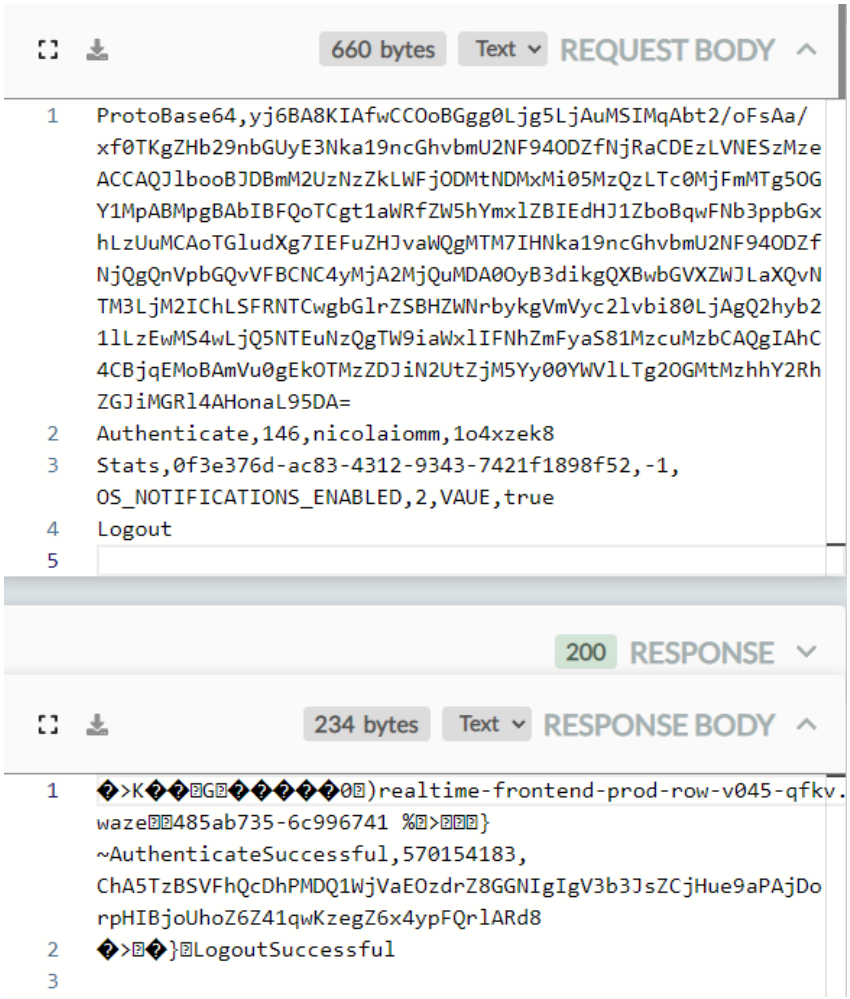


Figure 4.17. POST request and response on */rtserver/distrib/static*.

The first line is a string of the format *"ProtoBase64, <code>"*. Clearly it refears to some Base64 encoded string containing some kind of information. Using the online tool **protobuf-decompiler** it is possible to use general protobuffer definitions in order to decode the informations in that code. It is not always guaranteed that this is a working apprach, but in this case the tool is able to decode the informations in it. As expected, since the user is not yet logged in, there are no informations on the user account. On the contrary we can find some device-related informations like *device_manufacturer*, *device_model*, *Android_version*, *user_agent*, *device_resolution*.

The other lines of the body have the same format: the first word is indicating some kind of command to execute on the server, while on the same lines are contained the information related to that command. Here we have *Authenticate*, *Stats*, *Logout*.
The **Authenticate** command, as explained before, is communicating to the server to enstablish a new session with the Waze server. The data sent with is a number, the profile name, and a code.
The **Stats** command will just some statistical information, like the app notification enabled.

The last **Logout** command is unexpected, we will understand it in the next section.

The response obtained from the server is again composed of different lines, each one containing a Protobuffer code, but this time it is binary encoded. Again thanks to the online tool *protofbuf-decode* we are able to interpretate the informations contained in it.
The first line contains first of all the *response_timestamp* (expressed in milliseconds since epoch) and the actual *server_name* resolving our request. Moreover it contains the response to the *Authenticate* command specified in the request. The response to that is *Authenticate-Successful*, followed by a number, and a token.
The second line simply contains the response to the *Logout* command. The response is *LogoutSuccessful*.

At the end of this phase the application has communicated device informations to the server application, it has authenticated the user retrieving a token, and logged out.

Login phase

Once obtained the token from the user authentication in the previous phase, the application is ready to complete the login. A new POST request is issed on the resource *rtserver/distrib/login*. The body request and response are reported below:

Figure 4.18. POST request and response on */rtserver/distrib/login*.

The body structure of the request is slightly different from the previous case.
The first line is identical, a string of the format *"ProtoBase64, <code>"*, with an encoded Base64 protobuffer containing information device-related.
The second line this time contains the command **Login**, followed by username, a code and a timestamp.
The third line contains again a **"ProtoBase64"** value. This time the information contained

in it is only one, that is the *device_uuid*. Probably at the moment of the login, the application is linking the device with the account to log in, so this information is needed.
The last line is a **SuggestNavigation** command, followed by the coordinates expressed in integers of *longintude* and *latitude*.

The response body obtained is much bigger than the one of the previous phase containing 4.8 kilobytes of informations encoded in a binary protobuffer. The data contained in the response body will also contribute to the user interface showed inside the application, in fact all the structures will be taken by the application and built at runtime into objects for the UI and settings of the application.
The first line of the body contains the authentication token, that will be used for the subsequent requests from this time on. Among the relevant informations we can find a section where all the user account informations are transmitted:

Figure 4.19. Account information retrieved from */rtserver/distrib/login* body response.

As you can see first name, last name, email, account creation timestamp and username are embedded in the protobuffer obtained in the response body.
Now that the application has completed the login phase obtaining the authentication code, the user is logged in and the application enters the post-login phase.

Post-login phase

In this phase the application only communicates through POST request on the resource */rtserver/distrib/command*. Every request generated in this phase has the same structure:

- **Authentication line**: This is the first line of the body request. It contains the authorization part of the request specifying *UserID* and *Authorization token*, and a number probably indicating the Waze version used.
  Looking more carefully at the authorization token through the tool *protobuf-decoder*,

it seems to be a ProtoBuffer structure containing some informations on the user. In particular it is composed of:

Field Number	Type	Content
1	string	s55t75xT0k86RIN6
2	varint	As Int: 1676544162 As Signed Int: 838272081
3	varint	As Int: 4196 As Signed Int: 2098
4	string	world
5	varint	As Int: 1409834064 As Signed Int: 704917032
6	varint	As Int: 1761892200 As Signed Int: 880946100
7	string	⌧v⌧H⌧ɜΣ⌧<sVZJ⌧⌧Q⌧⌧⌧

Figure 4.20. Authorization token composition.

In this structure is easy to denote a *code*, a *timestamp*, and other values meaningless at a first sight.

- **Command lines**: Every other line contained in the body requests will refer to a command that is being communicated to the Waze server. The structure of these lines is: *<command>, <parameter1>, ..., <parameterN>*.
- **Data lines**: Together with commands might be also transmitted entire data structure to the server. These data structure are encoded using the ProtoBuffer format. A line in the body request containing a data structure is composed of the string *ProtoBase64*, followed by the Base64 code relative to the ProtoBuffer string representing the data structure.

The body response for this kind of requests are again Protobuffers, but this time in simple binary format.

Right after the login phase, the application will automatically generate some POST requests, communicating client informations to the Waze service. Among these requests, two of them are particularly interesting from the point of view of the information shared.

1. *Location and Time Request*: This is a request composed of two single lines. The first one is the authentication line, while the other one is a ProtoBuffer structure. By analyzing the structure in the *protobuf-decoder* tool, the information contained in the structure are *location coordinates* and a *timestamp*.
2. *IP Address Request*: This is again a request composed of two lines. The authentication line and a ProtoBuffer structure containing the *IP Address* of the client connected to the Waze service.

Now that we know how requests and response are structured, some specific user actions are monitored while using the Waze application. More specifically the behaviour of the

application while the user is not directly using the application (idle process), and while the user is actively moving the map on the screen in the application.

Idle behaviour

As I explained, Waze is designed to assist the driver notifying it on the traffic news. The way in which the Waze application is developed is really similar to a real time client-server application, meaning that even when the user is not directly using the application the client will still be active and sending location information to the Waze servers.
The application has been investigating while active in foreground without any kind of user activity. Roughly every 2 minutes the application send a POST request containing the **At** command, followed by some **Stats** commands. This is the request and response captured in HttpToolkit:

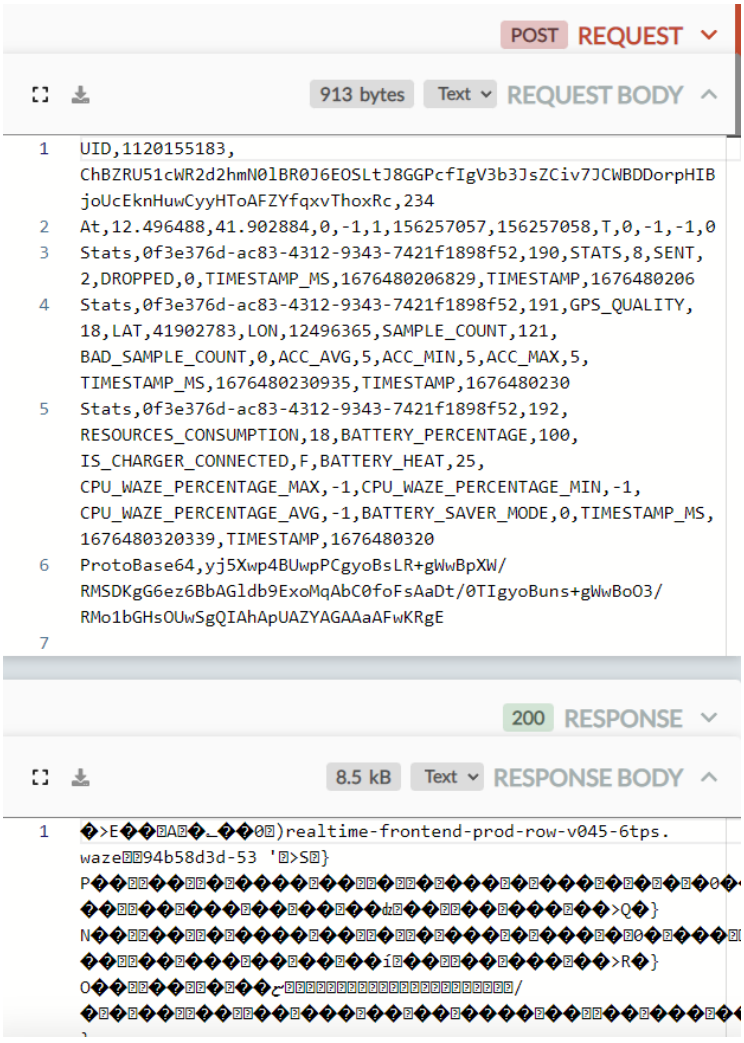


Figure 4.21. Idle request and response to */rtserver/distrib/command*.

The structure of the body request is the one described in the previous section.
The parameters passed to the *At* command are *longitude* and *latitude*, together with some

other numbers.
The parameters passed to the *Stats* commands can vary specifying *gps_quality*, *latitude*, *longitude*, *resource_consumption*, *battery_percentage*, *battery_heat*, *battery_save_modality*. The last line is a *ProtoBase64* containing a data structure. Analyzing the Protobuffer in the online tool *protobuf-decoder* it is a structure containing four pairs of longitude and latitude coordinates, with a timestamp and other integer values. Placing the four coordinates on a map we note that they are relative to the four location where the edges of the screen are met on the map of the Waze application. The picture below will simply explain this concept:

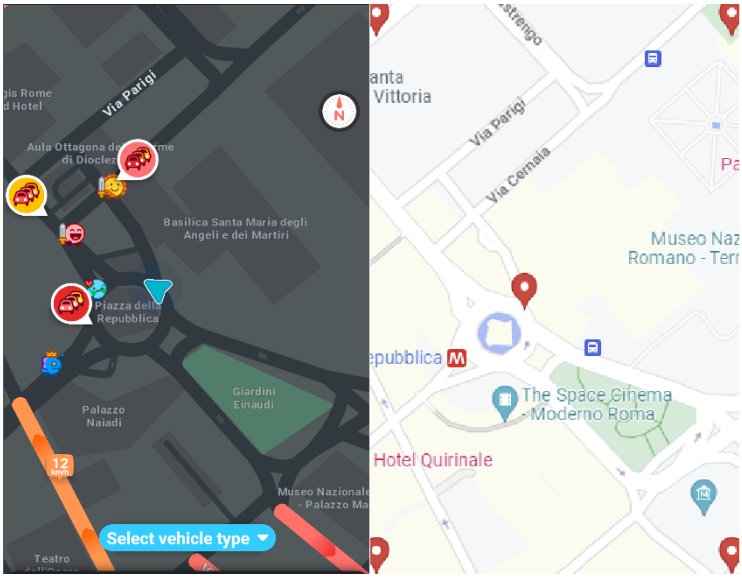


Figure 4.22. Location coordinates sent (explaination).

As response to this request, the server will sent in the body the data needed by the client to update the visualization of the information on the map, meaning that everything we see on the Waze application has to be somewhere in the body response.

For example the traffic information on *Via Torino* (left-bottom corner of the map) can be identified in the following line of the body response:

```
nAddRoadInfo,1803450860,32,2,0,Via Torino,Roma,,Via Nazionale,F,F,0,0,6,0,-1,
    T,-1,-1,-1,-1,F,1,41,422,,-1,1000
```

At the same way traffic informations and Waze users information are retrievable in the body. I will explain later what are the informations obtained on other users.

Moving the map

While actively using Waze, every interaction with the application will generate a request to the Waze server. When for example the user will drag the Waze map to check for near traffic information a POST request is issued to the server, containing the **MapDisplayed** command. The parameters passed are a list of longitude and latitude locations, together with a number. The syntax of this command has been decoded as follow:

```
MapDisplayed, <ScreenTopLeft_long>, <ScreenTopLeft_lat>, <ScreenTopRight_long
    >, <ScreenTopRight_long>, <ScreenBottomRight_long>, <
    ScreenBottomRight_lat>, <ScreenBottomLeft_long>, <ScreenBottomLeft_lat>,
    <ScreenCenter_long>, <ScreenCenter_lat>, <ZoomLevel>, <MapTopLeft_long>,
    <MapTopLeft_lat>, <MapTopRight_long>, <MapTopRight_lat>, <
    MapBottomRight_long>, <MapBottomRight_lat>, <MapBottomLeft_long>, <
    MapBottomLeft_lat>
```

Similarly to the *At* command, all the corners location that the device edges are forming on the visualized Waze map are sent. This time also an integer relative to the map zoom level is sent as parameter.

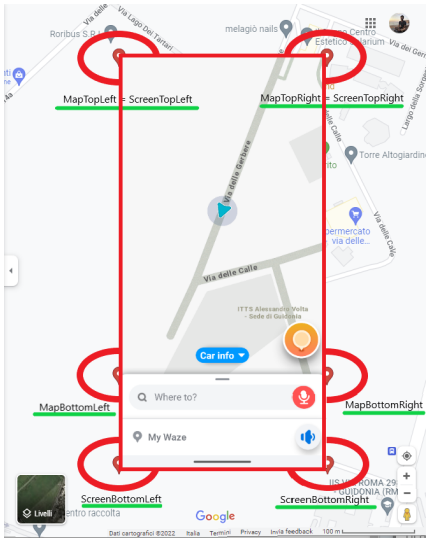


Figure 4.23. Location coordinates sent (explaination).

The data that the client obtain in the body response are equivalent to the one described in the previous section, containing informations relative to any waze user or traffic data visualized on the application map. Those informations are encoded using the ProtocolBuffer format and has been investigated in the following section.

Informations retrieved by Waze servers

We have already discussed two commands used by the Waze client application to retrieve data from the Waze server and show these informations on the screen.

Taking as example the body response received from the *MapDisplayed* command, we can define the following data structure for the whole ProtoBuffer data contained in each response:

- **Server_Info**: It contains server related informations like *timestamp*, *server_name*, **request_sequence_number**.
- **Users_Info**: Here are contained informations on the other Waze users. Basically the server will send information on only the users inside the portion of the map showed in

the application, for this reason they are sent as parameters in the request. This is a list containing each Waze user in that portion of the map. Associated informations are: *UserID*, *longitude* and *latitude*.

- **Road_Info**: This is the section relative to the information on the roads near our location. We have already shown the example of *via Torino* in the previous section.

- **Alerts_Info**: Alerts appear as circles on the Waze map and denote an event that some other Waze users have signaled. Other users can write comments and like or dislike an alert. All the information alert-related are contained in this section.

Since in this study we are particularly interested in the user informations shared through the communication protocol let's analyze more in-depth the user-related informations.

Other users informations

As discussed in the previous section, there is a specific section *User_Info* of the Protocol Buffer structure received from the Waze servers as response for each request issued. Together with this portion of data, also the section *Alerts_Info* might contains others user informations that have interacted with some alert on the Waze map. For example if a user have commented an alert signaling a road strike, then these comments will be communicated in this section of the Protocol Buffer.

Dealing with the **User_Info** section of the Protocol Buffer, this is how the tool *protobub-decoder* tool will parse the data related to each other user in the map area delimited by the device edges:

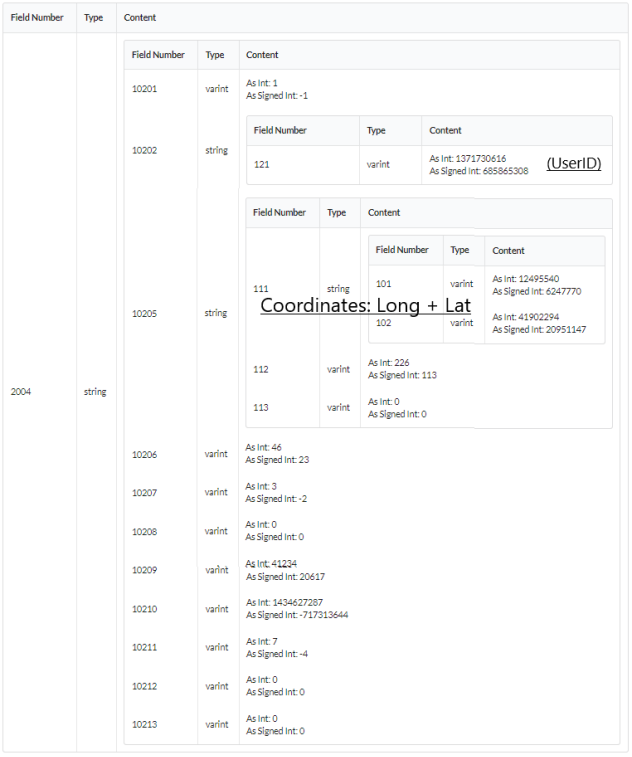


Figure 4.24. Other Waze users informations in ProtocolBuffer format.

Of course while decoding raw data, the tool is not able to understand the semantics of these value fields, we would need the *.proto* files in order to completely solve that question. Anyway some of these informations can be quite easily to understand. In this case we notice the *UserID*, and the *longitude* and *latitude* coordinates. Other values seems to be meaningless for the moment.

The same approach has been conducted for the **Alert_Info** section. For size issue the whole data structure is omitted, instead only the relevant fields are reported. In this case the alert was related to a closed lane.

- *Alert_ID*: 1818310871
- *Alert_Location: Longitude + Latitude*: 12770914; 41922559
- *Alert_name*: Restringimento di carreggiata con senso unico alternato
- *Creator Username*: miole67
- *Creation alert Timestamp*: 1666087200
- *Alert Comments*:
  - *Related alert_ID*: 1818310871
  - *Username*: LupiSolitariEventi
  - *Comment string*: Sono 3 anni che c'è questo semaforo!!!

– *Comment Timestamp*: 1666223821

So when retrieving data from the Waze server, in the *Alert_Info* section there are only the usernames relative to the alert creator and the users commenting that specific alert.

UserIDs and Usernames

Once decoded the actual informations involved in the communication protocol an additional investigation is needed in order to understand if those informations are immutable for every Waze users. In fact if the user can everytime change its username, the string is not anymore an identifier for the user. The same concept is valid for the userIDs. Generally an userID is assigned from the service and fixed for the whole duration of the usage of the service.

In the Waze application case the username is decided at the moment of the registration of a new account. Anyway at any moment the user is able to change its username from the settings panel of the application.
At the same way, the userID assigned to each user at the moment of the login (described in the above section) is refreshed at every login phase. Meaning that when the application is closed (not even running in background), and reopened, the login phase is executed again, and a new userID is assigned to the user.

4.4.2 Results

The results of the investigation on the Waze application shows that some important informations on the users are shared in the communication protocol. In any case all the data is encrypted using the HTTPS protocol. Moreover the ProtocolBuffer format is adopted in order to let the decoding of the data more trivial.

Private informations

The presence of a Man-In-The-Middle attacker reveals that it is possible to retrieve information on:

- Device informations: *device_manufacturer*, *device_model*, *user_agent*, *Android_version*, *device_resolution* (see Pre-Login Phase section).
- Account and User informations: *first_name*, *last_name*, *email*, *account_creation_timestamp*, *username*, *location_coordinates*, *ip_address* (see Login Phase section).
- Other users information: *userID*, *username*, *location_coordinates* (see User informations section).

Regarding to the *userID*, *username*, informations related to the other Waze users anyway it has to be precised that they might not contribute to the deanonymization of specific Waze users, since those values can change from execution to execution.

In any case once fixed the *userID* field for the whole session duration, it is possible to exploit these informations for **geofencing** purposes. It is possible in fact to retrieve userIDs and location coordinates in a virtual perimeter around the device position.

4.5 Application case: GoogleMaps

4.5.1 Study detail

4.5.2 Sensitive data

4.5.3 Other vulnerabilities

Chapter 5

Conclusion

Bibliography

[1] IPS Communication Intelligence Solutions
https://www.ips-intelligence.com/

[2] European Commission - What is personal data
https://commission.europa.eu/law/law-topic/data-protection/reform/what-personal-data_en

[3] Android Developers - Security with network protocols
https://developer.android.com/training/articles/security-ssl

[4] Cloudflare - TLS Handshake
https://www.cloudflare.com/it-it/learning/ssl/what-happens-in-a-tls-handshake/

[5] Okta Developer - Certificate Validation
https://developer.okta.com/books/api-security/tls/certificate-verification/#tls-certificate-verification

[6] Digicert - What is certificate pinning
https://digicert.com/blog/certificate-pinning-what-is-certificate-pinning

[7] Scotthelme.co.uk - Using security features to do bad things
https://scotthelme.co.uk/using-security-features-to-do-bad-things/

[8] Appmattus Medium - Android Security SSL Pinning
https://appmattus.medium.com/android-security-ssl-pinning-1db8acb6621e

[9] Protocol Buffers Docs - Overview
https://protobuf.dev/overview/

[10] gRPC.io - Docs
https://grpc.io/docs/

[11] IETF - RFC 9000
https://datatracker.ietf.org/doc/html/rfc9000

[12] IETF - RFC 9114
https://datatracker.ietf.org/doc/html/rfc9114

[13] GitHub - rootAVD
https://github.com/newbit1/rootAVD

[14] Android Developers - Android Debug Bridge
`https://developer.android.com/studio/command-line/adb`

[15] Http Toolkit - Docs
`https://httptoolkit.com/docs/`

[16] PortSwigger - Configuring Android device to work with Burp Suite
`https://portswigger.net/burp/documentation/desktop/mobile/config-android-device`

[17] Frida.re - Docs
`https://frida.re/docs/home/`

[18] GitHub - Objection Docs
`https://github.com/sensepost/objection`

[19] GitHub - GDA
`https://github.com/charles2gan/GDA-android-reversing-Tool`