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	Co-Advisors
Prof. Leonardo Querzoni	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*
- *Akcknowledgments*
- *Collaboration with IPS*

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 are we doing it, or how are we doing it.

Going through the years it is becoming much more common to hear about the terms *data breaches* or *data leak* on a multitude of online applications. The whole amount of data that users has put in their account can be exposed to leak 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 and phone numbers, and so on. All of these data are private informations, maybe the user does not like to share to everyone else, but just with the system on order to get the customized experience the application is offering.

Mobile applications handle private data of millions of different 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.

1.1 Personal Data

The concept of personal data is something I would like to make clear. There are multiple definitions and laws defining the scope of this kind of informations. As stated by the European Commission [1]:

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.

1.2 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.

A mobile application might share private informations with or without our accord in different moment while using it. For instance our contact list might be shared while using an application when synchronizing our friend list.

Goal is therefore ensuring there is no leak of private data for the whole activity and at any moment for a specific application.

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.

1.3 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 our personal data.

Where the network sniffing offered a spotlight for some in-depth study, a **static analysis** has been adopted going through an examination of the code of the application. Static analysis on an Android application 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 like symbols stripping combined with the high number of classes, it is sufficient to hide something very interesting, for instance the method used to compute the *Authorization header* in an HTTP request.

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

1.4 Collaboration with IPS

Da definire ...

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 [2] 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.

From the performance point of view, the whole process of the TLS hand-

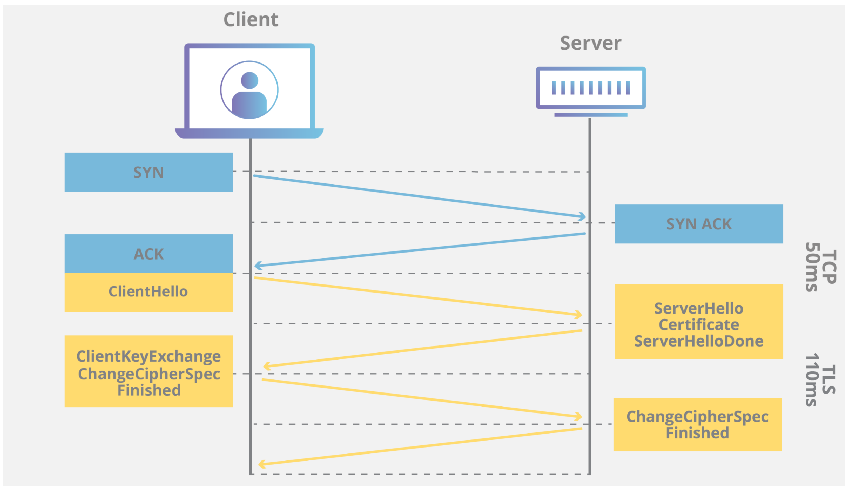

Figure 2.1. TCP Handshake and TLS Handshake communication flow needed to set up a secure connection. [3]

shake 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.

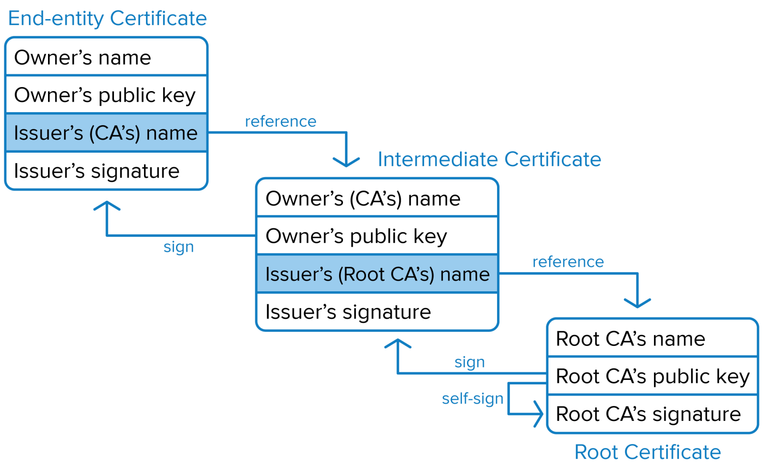

Figure 2.2. TLS Certificate verification chain. [4]

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 [6]). 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 [7]. 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[8], 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:

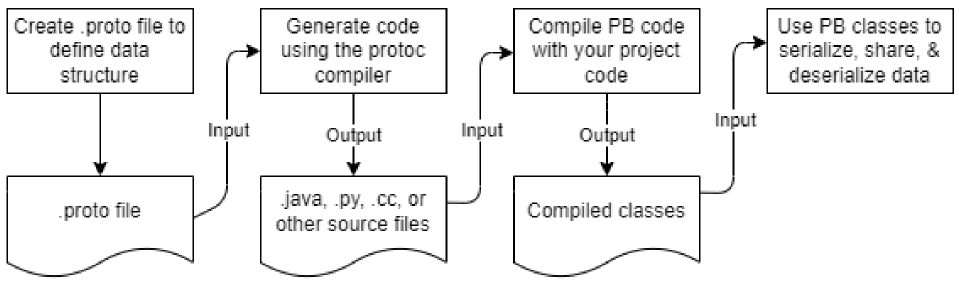
Create .proto file to define data structure
.proto file
Input
Generate code using the protoc compiler
Output
.java, .py, .cc, or other source files
Input
Compile PB code with your project code
Output
Compiled classes
Input
Use PB classes to serialize, share, & deserialize data

Figure 2.3. Procol Buffer workflow [8]

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 [9] 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.

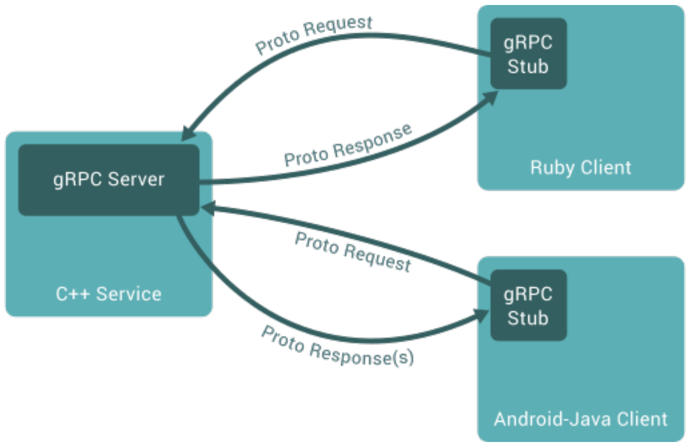

Figure 2.4. gRPC workflow [9]

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* [10].

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:

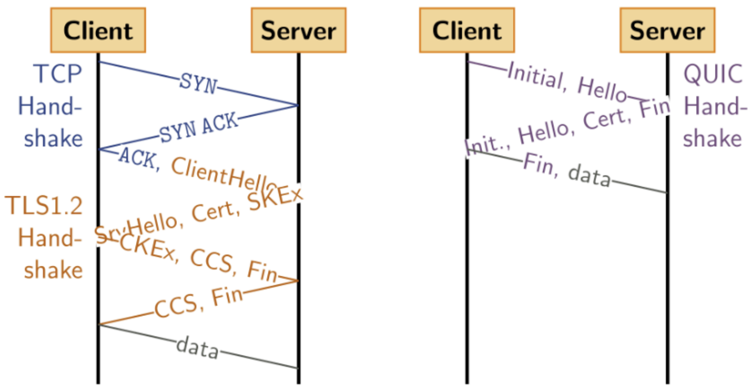

Figure 2.5. Comparison between TCP+HTTPS and QUIC handshakes

Moreover, as said above, UDP does not handle loss recovery, meaning that QUIC is also 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 *Hyper-Text Transfer Protocol* in *RFC 9114* [11]. 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 [12]. 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 [13].

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[14] 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 [15]:

- 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[16].
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[17].

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

4.1 Case: IlMeteo

4.1.1 General description

4.1.2 Study detail

4.1.3 Sensitive data

4.1.4 Other vulnerabilities

4.2 Case: Pacer

4.2.1 General description

4.2.2 Study detail

4.2.3 Sensitive data

4.2.4 Other vulnerabilities

4.3 Case: Waze

4.3.1 General description

4.3.2 Study detail

4.3.3 Sensitive data

4.3.4 Other vulnerabilities

4.4 Case: RadarBot

4.4.1 General description

Chapter 5

Conclusion

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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