Breaking best practice protection of the TLS protocol in an Android environment

Author
C.A.L. Mandjes
Master student Radboud University
c.mandjes@student.ru.nl

Supervisor
F. de Korte
Northwave Red team
frank.d.korte@northwave.nl

Supervisor
Prof. dr. E. Verheul
Radboud University
e.verheul@cs.ru.nl

Second reader
Dr. ir. E. Poll
Radboud University
e.poll@cs.ru.nl
Abstract
Northwave’s Red team performs penetration testing on Android applications for its customers. As part of this test, the tester analyzes the data which is being transmitted between the application and its backend server. However, many applications are using the Transport Layer Security (TLS) protocol, which is an encryption protocol, with the certificate pinning technique implemented. In order to set up a man-in-the-middle attack, Northwave was using a tool called the Android-SSL-Trustkiller. This tool disables the certificate pinning technique so that a man-in-the-middle position can be deployed.

However, the Android-SSL-Trustkiller is not working when the targeted Android application is using the SafetyNet Attestation service. Therefore, the goal of this research project is to find a new way to break the protection of the data sent in a TLS session, with certificate pinning and SafetyNet Attestation service implemented, in an Android environment. The SafetyNet Attestation service is an anti-abuse system that focuses on the integrity of the Android application and the Android device which it is running on. As part of the integrity checks on the Android device, it can detect whether a device is rooted. The Android-SSL-Trustkiller requires a rooted Android device; this is one of the reasons why this approach is not working when the targeted Android application is using the SafetyNet Attestation service.

This research project proposes a solution, called the Android-CertificatePinning-Killer, to bypass certificate pinning for an Android application by deploying code modification at runtime. The Android-CertificatePinning-Killer can bypass the following certificate pinning libraries: Android’s default library and the OKHttp library. Since the SafetyNet Attestation service can detect rooted Android devices, several experiments are performed to bypass this service. In these experiments, the two root implementations, system and systemless root, are assessed to bypass the SafetyNet Attestation service. Based on the performed experiments, it can be concluded that the rooted OnePlus 3T Android device could not be detected by SafetyNet’s device integrity checks when it is using systemless root.
Contents

1 INTRODUCTION ............................................................................................................. 7
  1.1 CONTEXT .................................................................................................................. 7
  1.2 PROBLEM DESCRIPTION ......................................................................................... 7
  1.3 RESEARCH QUESTION ......................................................................................... 8
  1.4 APPROACH ............................................................................................................. 8
  1.5 OUTLINE ................................................................................................................. 9

2 TRANSPORT LAYER SECURITY PROTOCOL .............................................................. 11
  2.1 INTRODUCTION .................................................................................................. 11
  2.2 HISTORY .............................................................................................................. 11
  2.3 PROTOCOLS ........................................................................................................ 12
    2.3.1 TLS Record protocol ...................................................................................... 12
    2.3.2 TLS Handshake protocol ............................................................................ 14
    2.3.3 TLS Change Cipher Spec protocol ............................................................ 17
    2.3.4 TLS Alert protocol ...................................................................................... 18
    2.3.5 TLS Application Data protocol ............................................................... 18
  2.4 SECURITY CONTROLS ........................................................................................ 19
    2.4.1 Confidentiality .............................................................................................. 19
    2.4.2 Integrity ........................................................................................................ 19
    2.4.3 Authentication ............................................................................................. 19
    2.4.4 Forward secrecy ........................................................................................... 19

2.5 PROTOCOL WEAKNESSES .................................................................................. 20
    2.5.1 Cryptographic aspects ................................................................................ 20
    2.5.2 Version aspects ............................................................................................ 21
    2.5.3 Implementation aspects ............................................................................. 21

3 DIGITAL CERTIFICATES ............................................................................................. 24
  3.1 INTRODUCTION .................................................................................................. 24
  3.2 CERTIFICATE AUTHORITY ................................................................................ 24
    3.2.1 Chain of trust .............................................................................................. 25
    3.2.2 Baselines ....................................................................................................... 25
  3.3 X.509 STANDARD ................................................................................................ 27
    3.3.1 Certificate format ........................................................................................ 27
    3.3.2 Additional extensions and techniques ...................................................... 28
  3.4 CERTIFICATE VALIDATION ................................................................................ 30
    3.4.1 Certification path processing .................................................................... 30
    3.4.2 HTTP Public Key Pinning ........................................................................ 31
    3.4.3 Certificate pinning ...................................................................................... 32

4 CERTIFICATE PINNING IMPLEMENTATION ............................................................. 34
  4.1 INTRODUCTION .................................................................................................. 34
  4.2 ANDROID TLS IMPLEMENTATION .................................................................. 35
  4.3 ANDROID CERTIFICATE PINNING IMPLEMENTATION .................................. 36
  4.4 iOS IMPLEMENTATION ...................................................................................... 38

5 ANDROID SAFETYNET SERVICE ............................................................................. 40
  5.1 INTRODUCTION .................................................................................................. 40
  5.2 SAFETYNET ATTESTATION PROCESS ............................................................. 40
    5.2.1 Workflow overview ................................................................................. 41
    5.2.2 Workflow steps ......................................................................................... 42
    5.2.3 SafetyNet’s attestation payload .............................................................. 43
List of Figures

Figure 1: TLS protocol in the OSI model .......................................................... 12
Figure 2: TLS record processing [9] ................................................................. 12
Figure 3: TLS Handshake protocol steps [9] .................................................... 15
Figure 4: Chain of trust ................................................................................. 25
Figure 5: Northwave's digital certificate ........................................................ 27
Figure 6: Northwave's digital certificate's certification path ............................ 30
Figure 7: Example TLS connection implementation [58] .......................... 35
Figure 8: Example certificate pinning implementation with Android's default library [58] .... 37
Figure 9: Example certificate pinning implementation with the OkHttp library [72] .......... 38
Figure 10: SafetyNet Attestation workflow [6] ................................................. 42
Figure 11: Example SafetyNet's attestation payload ...................................... 44
Figure 12: Example SafetyNet Attestation implementation [6] .................... 46
Figure 13: TLSConnector home screen after startup ..................................... 47
Figure 14: TLSConnector SafetyNet Attestation process using Android's default library ...... 48
Figure 15: TLSConnector SafetyNet Attestation process using the OkHttp library ...... 48
Figure 16: TLSConnectorServer expected values SafetyNet's attestation check ............ 49
Figure 17: Example Frida code modification script ....................................... 51
Figure 18: Rooted Android device Frida process overview ............................. 52
Figure 19: Nonrooted Android device Frida process overview ......................... 53
Figure 20: Bypass Android's default library by hooking checkTrustedRecursive() .... 55
Figure 21: Bypass Android's default library by hooking init() ......................... 55
Figure 22: Bypass OKHttp library by hooking add() ...................................... 56
Figure 23: Information logging SafetyNet Attestation API by hooking attest() and getJwsResult() ................................................................. 57
Figure 24: Format try-catch block for code modification script ...................... 58
Figure 25: Android-CertificatePinning-Killer output in the console .................. 59
Figure 26: Android-CertificatePinning-Killer nonce response in Burp Suite ............. 59
Figure 27: Warning screen verifiedstate yellow [114] .................................... 62
Figure 28: Nexus 5X SafetyNet Attestation check ......................................... 65
Figure 29: Nexus 5X OEM unlocking option is disabled .................................. 65
Figure 30: Nexus 5X systemless root location .............................................. 66
Figure 31: Nexus 5X sbin folder .................................................................. 66
Figure 32: Nonrooted experiment with modified TLSConnector ...................... 66
Figure 33: TWRP home screen ................................................................... 68
Figure 34: TWRP adb sideload .................................................................... 68
Figure 35: LineageOS system root experiment SafetyNet Attestation check .......... 69
Figure 36: LineageOS system root location .................................................. 70
Figure 37: Stock Android systemless root experiment SafetyNet Attestation check .... 70
Figure 38: Stock Android systemless root location ....................................... 70
Figure 39: LineageOS systemless root experiment SafetyNet Attestation check .... 71
Figure 40: OnePlus 3T output Key Attestation Demo ..................................... 72
Figure 41: Android-CertificatePinning-Killer check response in Burp Suite ............ 73
List of Tables

Table 1: Examples TLS Alert protocol messages [9] ................................................................. 18
Table 2: X.509 extension Key usage values [43]................................................................. 28
Table 3: Examples SafetyNet's values for ctsProfileMatch and basicIntegrity [6] ............... 45
1 Introduction
This chapter describes the introduction of this thesis. This thesis is written as part of a research project which is performed at Northwave between the 1st of November 2019 and the 11th of May 2020. Northwave is a company that is based in Utrecht, the Netherlands, and is specialized in cyber security operations [1].

In section 1.1 follows a brief summary of the protocols and techniques which are being discussed in this thesis; those topics are discussed in more detail later in this thesis. With the discussed protocols and techniques, section 1.2 focusses on the problem description for this thesis. The problem description comes primarily from Northwave and was molded in a research question which is further described in section 1.3. This section also defines the scope of this project. Section 1.4 describes in more detail the approach of this research project. Finally, section 1.5 describes the outline of the rest of this thesis.

1.1 Context
In the interconnected world of today, confidential information is transmitted daily over the air or cable. The information being transmitted can be a high value and attackers may be interested in intercepting or even tampering with this data. In order to prevent this, many (encryption) protocols have been developed over the years.

The Transport Layer Security (TLS) protocol, and its predecessor the Secure Socket Layer (SSL) protocol [2], are communication encryption protocols [3, 4]. The TLS protocol achieves, when properly configured, privacy, data integrity and authenticity for information exchanged between two communicating parties. In order to achieve authentication for the server, and optionally the client, the agent sends its digitally signed certificate to the party which it is communicating with. This certificate is proof of identity for an entity. Furthermore, a certificate also contains cryptographic material which can be used to set up an encrypted channel between the two communicating parties.

The party which receives this certificate is required to verify the certificate on its validness. By validating the certificate, the communicating party can be ensured that it is communicating with a legitimate agent. The so-called certificate pinning is a technique that can be used to assess the validness of the provided digital certificate. By using certificate pinning, the receiving party expects a specific certificate for a specific host. In case the provided digital certificate is not matching the expected (pinned) certificate, the authentication will fail, and the connection will be terminated. Certificate pinning needs to be implemented in the application logic and can prevent man-in-the-middle attacks.

1.2 Problem description
Northwave’s Red team performs penetration testing on Android applications for its customers. By performing a penetration test, the tester searches for possible vulnerabilities and exploits in these mobile applications. As part of this test, the tester also analyzes the data which is being transmitted between the application and its backend server. However, many applications which are being tested, are using the TLS protocol with the certificate pinning technique implemented.
Therefore, a tool called the Android-SSL-Trustkiller was being used to break the protection of the TLS session(s). This was achieved by breaking the certificate pinning technique of the Android application and deploying a man-in-the-middle position [5]. By using this tool, together with a test environment, it was possible to analyze and alter the communication between the application and its backend server in plain text. This gives the tester a better view of the application's infrastructure and possible vulnerabilities.

However, it is not possible to use the test environment with the Android-SSL-Trustkiller, when the Android application is using the SafetyNet Attestation service [6]. The SafetyNet Attestation service performs an integrity check on the Android device; this check is, according to Northwave, not being passed when the Android-SSL-Trustkiller is used. When the Android-SSL-Trustkiller is targeting an Android application that uses the SafetyNet Attestation service, the device integrity check will not pass. This results in that the Android application will not work and therefore the tester won't be able to analyze and alter the communication. This is crucial as part of the test.

In order to solve the problem described above, Northwave asks its customer to disable the SafetyNet Attestation service when it is performing a penetration test on an Android application. However, when the Android application is altered, it is unrepresentative compared to its original release. Therefore, Northwave is looking for a new approach to break the TLS protocol in an Android application with certificate pinning and the SafetyNet Attestation service implemented.

1.3 Research question
The focus of this research project is to find a way to break the protection of TLS sessions, with certificate pinning and the SafetyNet Attestation service implemented, in an Android environment. In order to do this, deployment of an active attack on the encrypted communication will be discussed in this thesis. The goal is to observe and alter the encrypted communication between the Android application and its backend server in plain text. Additionally, it is allowed to use a rooted Android device to solve this problem [7]. To achieve the goal described above, this thesis will be led by the following research question:

“Can the protection of the data sent in a TLS session, with certificate pinning and SafetyNet Attestation service implemented, be broken in an Android environment?”

1.4 Approach
This study starts with a literature review about the TLS protocol, and its predecessor the SSL protocol, and digital certificates. This gives a better understanding of how these protocols work, and how digital certificates are used to achieve authenticity; this also includes an analysis of how digital certificates can be verified using techniques such as certificate pinning.

Since this research project is focused on an Android environment, the implementation of the TLS protocol, certificate pinning and the SafetyNet Attestation service, will be analyzed in more detail. This includes an analysis of libraries to implement certificate pinning. Furthermore, to set up a test environment and to get a better understanding of these techniques; an Android test application, with a simple backend server, will be developed during this research project. In this Android application, the TLS protocol, popular certificate pinning libraries and the SafetyNet Attestation service will be implemented. Later in this study, this Android application can be used to analyze whether there a technique to break and alter the data being transmitted in TLS session(s).
Once there is a clear view of how these techniques can be implemented in an Android environment, several experiments will be performed to break the encrypted communication. Those experiments will be assessed using the developed Android application. Firstly, there will be an analysis of how the Android-SSL-Trustkiller is patched by the SafetyNet Attestation service. Secondly, based on the outcome of the analysis, there will be a reflection about alternative approaches to break and alter the encrypted communication in an Android application which has the SafetyNet Attestation service implemented. This can be achieved by deploying an active attack. An active attack may result in a man-in-the-middle position in which the attacker can be read and alter the encrypted communication in plain text.

In order to obtain the required information to answer the research question for this project, the following sub-research questions need to be answered:

1. How does the TLS protocol achieve confidentiality, data integrity and authenticity of the information exchanged between two communicating parties?
2. What are the known weaknesses of the TLS protocol?
3. How does certificate pinning work?
4. How can the TLS protocol be implemented in an Android environment?
5. How can certificate pinning be implemented in an Android environment?
6. What are the security checks the SafetyNet Attestation service is performing?
7. How has the SafetyNet Attestation service patched the Android-SSL-Trustkiller?
8. Can certificate pinning and the SafetyNet Attestation service be bypassed in order to break the TLS protection?

1.5 Outline
The remainder of this thesis is organized as follows. Chapter 2 provides a literature review of the TLS protocol. This chapter describes fundamental information about the TLS protocol with respect to its (sub)layers and (sub) protocols. In addition, it describes the known weaknesses of the TLS protocol. Chapter 3 provides a literature review of digital certificates. This chapter describes fundamental information about what certificates are and certificate validation techniques such as certificate pinning. Chapter 4 provides a literature review of certificate pinning in an Android environment. In the introduction of this chapter, fundamental information about the Android operating system is shared. Next, chapter 4 provides information about how the TLS protocol and certificate pinning can be implemented in an Android environment, followed by a brief discussion on how the TLS protocol can be implemented in an iOS environment. Chapter 5 provides a literature review of the Android SafetyNet service. This chapter mainly focusses on the SafetyNet Attestation service, it provides detailed information about how this service works and how it can be implemented in an Android environment.

In chapter 6, information gathered from previous chapters will be used in practice; this results in a test environment to assess the experiments in chapters 7, 8 and 9. This test environment consists of an Android application and backend server which supports certificate pinning and the SafetyNet Attestation service. Chapter 7 is dedicated to an experimental analysis that focusses on bypassing certificate pinning in an Android environment. This chapter performs an analysis of why the Android-SSL-Trustkiller is patched, it introduces an alternative framework and describes experiments to bypass certificate pinning. The output of this chapter is a code modification script, called the Android-CertificatePinning-Killer, which works with the Frida framework. Chapter 8 is dedicated to an experimental analysis that focusses on bypassing the SafetyNet Attestation service. Since this service includes root detection, this chapter starts with an analysis of the rooting topic. At the end of this chapter follows several experiments with as goal to bypass the SafetyNet Attestation service. The output of this chapter is a method to bypass the SafetyNet Attestation service. Chapter 9 combines the
results of chapters 7 and 8; bypassing certificate pinning and the SafetyNet Attestation service. These results are combined and examined in the wild using several Android applications from the Google Play store.

Chapter 10 describes the conclusion of this thesis. In this chapter, the research question and sub-research questions are answered. Chapter 11 provides a discussion and information for future research, based on the results of this thesis; those results are mainly described in chapters 7, 8 and 9. And last but not least, chapter 12 is dedicated to acknowledging everybody who was involved in this research project.
2 Transport Layer Security Protocol

This chapter describes the Secure Socket Layer (SSL) and Transport Layer Security (TLS) protocol in detail; this information is required to get a better understanding of these protocols. This chapter starts with a brief introduction about the SSL and TLS protocols, this is described in section 2.1. Since there are various versions of the protocol available, section 2.2 gives a brief description of which ones are developed over the years. The SSL and TLS protocols comprise several sub-protocols, those are explained in section 2.3. Furthermore, several security controls can be achieved, depending on its configuration, with the SSL and TLS protocols; these security controls are described in section 2.4. Also, over the years many attacks and weaknesses are noticed. Section 2.5 gives a summary of some of those attacks and weaknesses to get a better understand of how the SSL and TLS protocols could be broken over the years.

2.1 Introduction

The Transport Layer Security (TLS) Protocol, and its predecessor Secure Sockets Layer (SSL) protocol\(^1\), are protocols designed to establish a secure channel between two parties \[8\]. The TLS protocol aims at confidentiality, integrity and authentication between the communicating parties. The TLS protocol supports many configurable features, which can be set during the initialization of the TLS connection.

The TLS protocol can run over any transport layer protocol. However, the Transmission Control Protocol (TCP) is the most commonly used transport layer protocol. Furthermore, the TLS protocol comprises several other protocols, those will be discussed in section 2.3.

2.2 History

The SSL protocol was originally developed by Netscape which was used to ensure secure transactions of customer’s data. The goal of this protocol was to ensure confidentiality, integrity and authenticity for its transactions. The first public version of the SSL protocol was version 2.0; however, this version was quickly replaced by SSL protocol version 3.0 due number of discovered security flaws \[8\].

Since Netscape had the ownership over the SSL protocol, the Internet Engineering Task Force (IETF) standardized a new version of SSL protocol version 3.0, the upgraded version became known as TLS protocol version 1.0. During the course of the years, IETF continued iterating on the protocol for security improvements. This resulted in TLS protocol versions 1.1, 1.2, and the most recent version 1.3.

2.3 Protocols

The TLS Handshake protocol, TLS Change Cipher Spec protocol, TLS Alert protocol, and the TLS Application data protocol, which are capsulated in the TLS Record protocol, comprises the TLS protocol. The TLS protocol is positioned between the Transport layer and the Application layer \[9\]. Figure 1 illustrates the TLS protocol, and its sub-protocols, in the OSI model.

Furthermore, this section gives a brief description of TLS’s sub-protocols to get a better understanding of how a TLS session is established and how the encrypted communication starts between the communicating parties.

---

\(^1\) The term “TLS” will be used to refer to both “TLS” and “SSL” protocol in the rest of this thesis; unless, a protocol specific version specification will be addressed.
2.3.1 TLS Record protocol

The TLS Record protocol is dedicated to encapsulating the data of TLS's sub-protocols. This protocol splits the data into manageable pieces called fragments, those fragments are processed individually. With the TLS Record protocol, the fragments are optionally compressed and cryptographically protected according to the compression method and cipher suite which the parties agreed on during the TLS Handshake protocol [9]. The TLS Handshake protocol is explained in section 2.3.2.

The fragmentation, compression, and cryptographic protections leads to the following data structures: TLSPlaintext, TLSCompressed, and TLSCiphertext. Once the processing is finished, the TLSCiphertext will be appended to the TLS record header to form the TLS record which will be sent to the communicating party. The TLS record header comprises four fields: type, version, length, and a fragment field. TLS record processing is illustrated in figure 2. The TLS record processing is performed in and authenticate-then-encrypt, an analysis of each step follows below.

Figure 2: TLS record processing [9]
2.3.1.1 TLS record processing
This section describes the steps which are taken for each fragment of the TLS Record protocol. The literature being used for this section is the book called SSL and TLS: Theory and Practice [9].

Fragmentation
The first step of the TLS Record protocol is splitting the message into fragments. Those fragment blocks are packed into a TLSPlaintext structure.

Compression
The second step of the TLS Record protocol is to compress the TLSPlaintext structure according to the compression method, this method is optional by default. However, this method can be set during the TLS Handshake protocol. The compression of the TLSPlaintext structure results in a TLSCompressed structure. If there is no compression method set in the TLS session state, the TLSPlaintext and TLSCompressed are identical.

Cryptographic protection
In the third step of the TLS Record protocol is to encrypt the TLSCompressed according to the cipher suite of the session state. This results in the TLSChiphertext, which is the cryptographically protected output together with message authentication. There is always an active cipher suite, but it is initially it is set to TLS_NULL_WITH_NULL_NULL, which does not provide encryption nor message authentication. The actual cipher suite of the TLS session state can be set during the TLS Handshake protocol negotiation. The message authentication is achieved with a Hash-based Message Authentication Code (HMAC) [10], which uses the hash algorithm of the cipher suite. For a more detailed explanation about HMAC can be found in the book SSL and TLS: Theory and Practice [9].

In the TLS protocol, the TLSChiphertext is by default authenticate-then-encrypt (AtE). However, it has been shown by Bellare and Namprempre [11], and Krawczyk [12] that AtE is no longer regarded as secure. Therefore, RFC7366 [13] specifies an option in the TLS protocol to perform encrypt-then-authenticate (EtA) instead. This is only supported if both, the client and server, support this and it is negotiated by both parties in the ClientHello and ServerHello message of the TLS Handshake protocol. Furthermore, another way to tackle this weakness is specified in RFC8446 [4], which is the RFC for TLS protocol version 1.3, which describes that only Authenticated Encryption with Associated Data (AHEAD) algorithms are permitted for this protocol version [14].

TLS Record header
In the fourth step, which is the final step in the TLS Record protocol, a TLS Record header is appended to the TLSChiphertext structure. This results in the final structure, which is the TLS record. The TLS Record header comprises the following three fields:
- **Type**: Refers to the higher-layer TLS protocol. There are four predefined values: TLS Handshake protocol, TLS Change Cipher Spec Protocol, TLS Alert protocol, and TLS Application Data Protocol.
- **Version**: Version of the SSL/TLS protocol in use.
- **Length**: Byte-length of the following higher-layer protocol messages which are transmitted in fragments.
Once this step is succeeded, it is ready to be sent to the recipient in a TCP segment. Note that when multiple TLS Records are sent to the same recipient, these records may be sent together in a single TCP segment. Furthermore, the header fields are not authenticated nor encrypted.

2.3.2 TLS Handshake protocol
The TLS Handshake protocol is layered on top of the TLS Record protocol. With the TLS Handshake protocol, the client and server can authenticate themselves and negotiate the cipher suite and compression method for the TLS session.

The goal of the TLS Handshake protocol is to establish a TLS session. With the TLS Handshake protocol, the communicating parties establish an asymmetric cipher. This cipher is called a session key and is used to encrypt the messages being transmitted for that session once it is established. The cipher suite which is used to establish the session key determines how secure the channel is and what security properties are assured. Furthermore, in the TLS Handshake protocol authenticity can be achieved. This is done by the server, and optionally the client, which proves its identity using a digital certificate; digital certificates are discussed in more detail in chapter 3. The steps performed in the TLS Handshake protocol are illustrated in figure 3; those steps are as follows [8, 9, 15]:

1. Client sends the preferred protocol version, random number, optionally session id, supported cipher suites and compression methods
2. Server sends the protocol version, which is lower or equal to the received version, random number, optionally a session id, decision for cipher suite and decision for the compression method
3. Server optionally sends its certificate along with the certificate’s chain
4. Server optionally sends parameters which are needed for the key exchange
5. Server optionally sends a request for the client to authenticate with a certificate
6. Server sends a message to indicate it is finished with sending messages
7. Client optionally sends its certificate; this only happens when it received a certificate request from the server
8. Client sends parameters, encrypted with server’s public key, which are needed for the key exchange
9. Client optionally sends a message, signed with its private key, to prove that it possesses the certificate’s public key
10. Client sends an indication that it is ready to communicate with the session key
11. Client sends an indication, encrypted with the session key, that the handshake was successful
12. Server sends an indication that it is ready to communicate with the session key
13. Server sends an indication, encrypted with the session key, that the handshake was successful

If one of the steps above fails, the TLS session will not be established, and the connection will be terminated.
2.3.2.1 Protocol messages
This section gives a detailed description of the most important details of the messages being sent during the TLS Handshake protocol. Those messages are illustrated in figure 3. This section is based on the book SSL and TLS: Theory and Practice [9].

1. *ClientHello* message
The *ClientHello* message is the first message that is sent by the client to the server in order to start the TLS handshake negotiation. In this message, the client sends the highest supported protocol version in the *client_version* field. In addition, the client also sets a client-generated random value. This value is stored in the *random* field. This value, together with a similar value provided by the server, provides input for later cryptographic computation. The *random* field is followed by the *session_id_length*. If the client wants to resume a connection, this field is set with the length of the session id value. If that is not the case, the value of this field is set to zero. This field is followed by the *session_id* field which comprises the id of the previous session; for a new session, this field is empty.

The *session_id* is followed by the *cipher_suites* field. A cipher suite is a set of algorithms that includes a key exchange algorithm, encryption algorithm and message authentication (MAC) algorithm. The first bytes of the *suites* field comprises the number of supported cipher suites by the client, followed by the actual list of supported cipher suites. This list is sorted on preference, with its highest preference on the top of the list. The cipher suites in this list are linked by a 2-byte code referring to it. The last part of the message being sent is the *compression_methods* field. This field contains a list of supported compression methods, ordered on preference, by the client.
Furthermore, if the session_id is set, the cipher_suites and compression_methods lists must contain the same settings in the list as the session the client wants to resume.

2. ServerHello message

After the server received the ClientHello message, it is up to the server to make decisions based on what the client sent; this is done by sending a ServerHello message to the client. This message closely resembles the ClientHello message. The significant difference is that the server sends single decisions, based on the provided lists from the client, for the cipher suite and compression method.

The ServerHello message has the server_version field, which comprises its highest supported protocol version which is equal or lower than client_version of the ClientHello message. This field is followed by a randomly generated number which is set in the random field. The structure of this number is identical to the one which is provided by the client. However, the value itself must be independent and different from the client’s random number. The server may also include the session_id_length and session_id after the random field. If the session_id of the ClientHello message was not empty, then the server checks its cache for a match. If the server found a match and is willing to re-establish a connection, the server will respond with the same session_id as provided by the client. Otherwise, a new session_id will be generated by the server and set in session_id. The server can also omit this functionality by setting the value of session_id_length to zero and leaving session_id empty.

The session_id is followed by the cipher_suite field which contains the value of its decision for the cipher suite which is going to be used for the TLS session. This decision is based on the provided list by the client. Finally, after the cipher_suite follows its decision for the compression method. This decision is set in the compression_method field and is also based on the provided list by the client.

3 & 7. Certificate message

In most cases, the server, and optionally the client, authenticates itself with a digital certificate. Digital certificates are explained in more detail in chapter 3. With a digital certificate, the identity of an entity can be proven and therefore be authenticated. Entity’s digital certificate, and the corresponding certificate chain, is sent in a certificate message. The length of the certificate chain is set in the certificate_chain_length field. After this field, the certificate chain itself is comprised in the certificate_list field. This field contains all certificates required to form the certificate chain. The sender’s certificate is followed by a series of CA certificates sequentially upward until a root CA is reached.

4. ServerKeyExchange message

The ServerKeyExchange message is used to transmit parameters for the key exchange. Whether this message is required is depending on what cipher suite is used. If, for instance, RSA is used, the client can retrieve the server’s public key from the certificate message and encrypt the premaster secret with this key. Similarly, when Diffie-Hellman key exchange is used, the client can retrieve the desired parameters from the server’s certificate, and therefore this message is not needed.

In some other cases, such as with ephemeral or anonymous Diffie-Hellman key exchange, the ServerKeyExchange message is required to provide the client with additional cryptographic parameters to establish the session key. The actual structure and content of the ServerKeyExchange message depend on what key exchange algorithm is being used to perform the TLS handshake.
5. **CertificateRequest message**
Optionally the server can also request the client to authenticate. This is only possible when the server is not anonymous, hence the server authenticated itself with a Certificate message. The CertificateRequest message requests the client to authenticate with its certificate. The server requires the client to authenticate with a fixed set of certificate types, therefore the CertificateRequest message starts with a certificates_types field. This field stores the length of this list along with one or more single-byte values that indicate the accepted certificate types. The certificate_types is followed by the certificateAuthorities field. This field indicates the accepted CAs which are used to sign the client's certificate.

6. **ServerHelloDone message**
The ServerHelloDone message is sent by the server to indicate the client that it is done with sending the ServerHello message and its associated messages. The body of this message is empty.

8. **ClientKeyExchange message**
The ClientKeyExchange message is an important message to establish a secure channel between the client and the server. This message is sent by the client and it provides the server with client-side cryptographic material that is later used to calculate the session key. The format of the ClientKeyExchange depends on the key exchange algorithm which is being used for the TLS handshake.

9. **CertificateVerify message**
If the server requested the client's certificate with a CertificateRequest Message, the client provides its certificate with a Certificate message. However, the client still has to prove that it possesses the corresponding private key. Therefore, the client sends a CertificateVerify message to the server which is signed with the client's private key. The body of this message comprises the digital signature. The exact format depends on the cryptographic cipher of the client's certificate. Note that the server is required to verify this signature before proceeding.

10 & 12. **ChangeCipherSpec message**
The ChangeCipherSpec message is not directly part of the TLS Handshake protocol, but a TLS sub-protocol on its own. This message is used to indicate the other communicating party that the session key is established, and it is ready to communicate using this key.

11 & 13. **Finished message**
The finished message is the last message of the TLS Handshake protocol to finish the handshake. This message is sent immediately after the ChangeCipherSpec message. The finished message is the first message of this protocol which is encrypted with the negotiated session key. This message needs to be sent by both communicating parties of that session before proceeding.

2.3.3 **TLSChangeCipherSpec protocol**
The TLS Change Cipher Spec protocol is a protocol that consists of a single message. This message is compressed and encrypted under the current, hence not pending, connection state. The ChangeCipherSpec message is sent by the client and server to notify each other that it is ready to use the negotiated session key [3]. This message will be sent in the TLS handshake as illustrated in figure 3.
2.3.4 TLS Alert protocol
The TLS Alert protocol allows the communicating parties to exchange Alert messages. Those messages can be used to notify the other party when something goes wrong during the TLS connection. The Alert message starts with an alert_level field. This field comprises 1 byte, where value 1 stands for “warning” and the value 2 for “fatal”. When the receiver receives a message with alert_level of warning, it may decide at its discretion whether to treat this as a fatal or not. A message with alert_level fatal should be treated accordingly by the receiver by immediate termination of the connection. The second field of the Alert message is the alert_description. This field also comprises 1 byte, this byte refers to a specific situation about the actual alert. An Alert message can be sent anytime by the communicating parties, the receiver is obligated to process this message in a good manner by checking the alert_level and its situation description in alert_description. Furthermore, the available alert_description values are depending on the version of the SSL or TLS protocol [9]. Table 1 illustrates examples of TLS Alert messages by alert, code, and description.

<table>
<thead>
<tr>
<th>Alert</th>
<th>Code</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bad_record_mac</td>
<td>20</td>
<td>Receiver of a message notifies the sender that the received message has an incorrect MAC. This alert should always be fatal.</td>
</tr>
<tr>
<td>handshake_failure</td>
<td>40</td>
<td>Send when a party concludes that it was not able to negotiate an acceptable set of security parameters given the options available. This alert should always be fatal.</td>
</tr>
<tr>
<td>no_certificate</td>
<td>41</td>
<td>Used by the client to notify the server that it has no certificate available which satisfies the requirements of the server.</td>
</tr>
<tr>
<td>bad_certificate</td>
<td>42</td>
<td>Used to notify the other party that the received certificate was corrupt.</td>
</tr>
<tr>
<td>unsupported_certificate</td>
<td>43</td>
<td>Used to notify the other party that the received certificate is not supported.</td>
</tr>
<tr>
<td>certificate_revoked</td>
<td>44</td>
<td>Used to notify the other party that the received certificate is revoked by the issuing CA.</td>
</tr>
<tr>
<td>certificate_expired</td>
<td>45</td>
<td>Used to notify the other party that the received certificate is expired.</td>
</tr>
</tbody>
</table>

Table 1: Examples TLS Alert protocol messages [9]

2.3.5 TLS Application Data protocol
The TLS Application Data protocol is layered on top of the TLS Record protocol. The TLS Application Data protocol is used once the session key is established. This protocol feeds the message into the TLS Record protocol for fragmentation, compression, and cryptographic protection. The TLS Record protocol is then sent to the other communicating party which decrypts, verifies, decompresses, and reassembles the received message. The TLS Record processing is illustrated in figure 2. After the TLS Record header, the payload is encrypted with the session key, only authorized parties should be able to see it in plain text [9].
2.4 Security controls
As described in the previous section, during the TLS handshake negotiation, the client and server agree upon the usage of cryptographic information. The TLS protocol does not specifically specify what cryptographic components to use. This all depends on which protocol version is being used and the supported algorithms by the client and server. Not all combinations of algorithms support all the available security properties [4].

Also, another important note is that algorithms may be deprecated over time. Therefore, it is important that the server not accepts every algorithm to establish a connection. Below follows a brief overview of security properties supported by the TLS protocol, and what the prerequisites are.

2.4.1 Confidentiality
Confidentiality is a security property in which only authorized parties can read the message; hence, the attacker cannot see the message in plaintext. The TLS protocol uses a combination of asymmetric and symmetric cryptography to achieve this. Whether confidentiality is achieved depends on the strength of the session key, but also on strength of the asymmetric keys depending on whether forward secrecy is achieved [4].

2.4.2 Integrity
Integrity is a security property that is about that messages being sent cannot be altered without any notice. In the TLS protocol, this is achieved with a Message Authentication Code (MAC). The MAC is a tag, which is part of the message being sent, which can be used to detect any changes to the message content. In the TLS protocol a Keyed-Hashing Message Authentication Code (HMAC), which is a type of MAC, is used. HMAC uses a cryptographic hash function, in combination with the session key. The strength of the HMAC depends on the underlying hash function [4, 10].

2.4.3 Authentication
Authentication is a security property to verify a claim of identity. During the handshake, the digital certificate of the server, and optionally of the client, are transmitted. Those certificates can be used to verify the identity of the other communicating party. The certificate also contains the public key of the agent. However, with the TLS protocol, the parties are responsible to verify the validness of the provided certificate [4]. Therefore, certificate validation needs to be carefully implemented; section 3.4 discusses certificate validation in more detail.

2.4.4 Forward secrecy
Forward secrecy is a security property that is about that a short-term key cannot be recovered from the long-term secret key. In a typical TLS setup, the client sends a nonce, encrypted with the server's public key, to the server which is then used to derive the session key. However, when the server's private key is compromised, the attacker can reconstruct the session key to decrypt all the messages for that session; therefore, forward secrecy is not achieved. Whether the forward secrecy property is achieved depends on what key agreement algorithm is used to establish the session key [4, 16].
2.5 Protocol weaknesses

Over the years, various weaknesses and attacks have been noticed for the SSL and TLS protocols. This session will summarize some of those attacks and is mainly summarized from a paper by Meyer and Schwenk [17]. The goal of this section is to give a better overview of the weaknesses of the SSL and TLS protocols. The weaknesses and attacks which are described in this section are divided into the following topics:

- Cryptographic aspects
- Version aspects
- Implementation aspects

2.5.1 Cryptographic aspects

Cipher suite rollback

This attack is discussed in a paper by Wagner and Schneier [18] and applicable on the SSL protocol version 2.0. This attack aims at limiting the offered cipher suites by the client. The attacker, with a man-in-the-middle position, changes the ClientHello message by altering the list of cipher suites, to weak ciphers or even NULL-ciphers, before it forwards this message to the server. When the server is not properly configured, it will accept the proposed cipher. This attack was later fixed in SSL protocol version 3.0 by authenticating all the messages in the Handshake protocol. This is achieved by including a hash value of all the messages sent in a session, encapsulated in the client and server Finished message.

ChangeCipherSpec message drop

This attack is discussed in a paper by Wagner and Schneier [18] and applicable for the SSL protocol version 2.0. After the key exchange in the Handshake protocol, the communicating parties sends a ChangeCipherSpec message to switch to encrypted communication. For this attack, the attacker with a man-in-the-middle position drops this message sent from both parties. This results in that both parties might never start the encrypted communication. RFC 2246 [19], which is the RFC for TLS protocol version 1.0, describes a recommendation that both parties should always receive the ChangeCipherSpec message before it accepts the Finished message. This recommendation is a countermeasure against this attack.

Key exchange algorithm confusion

This attack is discussed in a paper by Wagner and Schneier [18] and applicable for the SSL protocol version 3.0. During the key exchange of the Handshake protocol the client and server exchange information to establish a session key. For this attack, the attacker, with a man-in-the-middle position, changes the ciphers in the ClientHello message and forwards the message to the server. When the server responds with a ServerHello message, the attacker changes the cipher in this message to one of the originally proposed ciphers by the client. This results in that the client is performing a key exchange for a different cipher than the server. For instance, the client is performing an RSA based key exchange, while the server is performing a Diffie-Hellmann key exchange. This creates confusion for both parties once the session is established. This attack is so far only strictly theoretical.

ECC-based key exchange algorithm confusion attack

This attack is discussed in a paper by Mavrogiannopoulos, Vercauteren, Velichkov and Preneel [20] and has the same approach as the key exchange algorithm confusion attack by Wagner and Schneier, which is described above. However, this attack is focused on all versions of the TLS protocol using the Elliptic-curve Diffie-Hellman algorithm. This attack is, just like the one of Wagner and Schneier, not feasible yet due to computational limitations.
**MAC does not cover padding length**
This weakness is discussed in a paper by Wagner and Schneier [18] and applies for the SSL protocol version 2.0. This weakness is regarding the Message Authentication Code (MAC). In the SSL protocol version 2.0, the MAC only covers the data and the padding. However, the padding length field is left out in plaintext. This may lead to a message integrity compromise, any information about the actual plaintext should be considered as useful for an attacker.

**Weaknesses through CBC usage**
This weakness is discussed in a paper by Vaudenay [21] and applies to the usage of Cipher Block Chaining (CBC) encryption. In this paper, Vaudenay describes attacks against CBC encryption. This type of encryption operates in blocks of a fixed length. In most cases, the plaintext does not fit in a multiple of those blocks. Therefore, the text needs to be padded. This leaves room for an attacker to tamper this padding and use the server as a decryption oracle.

According to Vaudenay, by special crafted messages, and using the server as a decryption oracle, the ciphertext can be decrypted without knowledge of the encryption key. This can be achieved by listening to the server's response. When an Alert message is received, the attacker knows that the padding is incorrect. Furthermore, the MAC does not hinder this attack in the SSL or TLS protocol. The SSL and TLS protocols process messages in an order of authenticate-then-encrypt. Hence, the MAC creation takes place before the message is padded for encryption. This attack is so far strictly theoretical but should not be ignored.

**Message distinguishing**
This attack is discussed in a paper by Paterson, Ristenpart and Shrimpton [22] and is applicable for all versions of the SSL protocol and TLS protocol versions 1.0 and 1.1; for TLS protocol version 1.2, it is only applicable when it is using protocol extensions. With this attack, the attacker can distinguish two messages which are sent over an encrypted channel. This attack only works when the message has a short length and is using an 80-bit truncated MAC.

The attack is based on a clever modification of the eavesdropped message so that the server either accepts the message or returns an Alert message. By this, the attacker can distinguish messages which were being sent. This attack is patched in TLS protocol version 1.2 since 80-bit truncated MACs are not supported anymore by default.

### 2.5.2 Version aspects

**Version rollback**
This attack is discussed in a paper by Wagner and Schneier [18] and applicable for the SSL protocol version 3.0. For this attack, the attacker requires a man-in-the-middle position. The goal for this attack is to downgrade the SSL protocol version 3.0, to version 2.0. This can be achieved by altering the ClientHello message to look like a ClientHello message of SSL protocol version 2.0. This would force the server to switch back to SSL protocol version 2.0. As a countermeasure, the SSL/TLS protocol version is included in the ClientKeyExchange message.

### 2.5.3 Implementation aspects

**The rise of timing attacks**
This attack is discussed in a paper by Brumley and Boneh [23] and applicable on RSA based SSL and TLS. The goal of this attack is to get the private key of the server. This can be done by observing the time differences of specially crafted ClientKeyExchange messages and receiving the Alert messages. The time between sending and receiving messages allows the
attacker to draw a conclusion about the RSA parameters which are being used. In this paper, the authors described that they were able to successfully attack the OpenSSL library.

Information leakage by the use of compression
This attack is discussed in a paper by Kelsey [24] and applies to all versions of the SSL and TLS protocols. Kelsey concludes in this paper that the process of compression of the plaintext can be used for a side-channel attack. Data compression is a step which requires processing time depending on its cryptographic material. This attack was not directly exploited in this paper but led to the development of Rizzo and Duong’s C.R.I.M.E tool [25].

ECC based timing attacks
This attack is presented at ESORICS by Brumley and Tuveri [26] and is applicable on ECDSA based TLS connections. ECC heavily relies on scalar implementations, such as point multiplication. From a formal point of perspective, there are various ways to make this time resistant. However, even by using those countermeasures, from an implementational point of view, those implementations can contain timing side-channel vulnerabilities. Furthermore, they found OpenSSL to be vulnerable to this attack.

Brumleu and Tuveri also combined this side-channel attack with the lattice attack of Howgrave-Graham and Smart [27] to recover secret keys. The author concluded that ECDSA signatures rely on scalar multiplication. This kind of signature is used in the ECDHE_ECDSA cipher suite. By measuring the time between the sent ClientHello message and the arrival of the ServerKeyExchange message, it is possible to draw conclusions about the cryptographic material of that session. This is because the ServerKeyExchange message contains a signature which is created on the fly using scalar multiplications.

Breaking DTLS
This attack is discussed in a presentation by AlFardan and Paterson [28] and is applicable for TLS that runs over UDP; this is also called DTLS. DTLS is slightly different from the regular TLS which runs over TCP. Since UDP is an unreliable transport protocol, there are two major differences:

- Absence of Alert messages
- Messages causing error are simply dropped, instead of connection abortion

This gives room for the attacker since that bad messages will not result in session invalidation. But also, the attacker is not notified whether the modified message is correctly received. The authors of this paper used an enhanced version of Vaudenay’s attack. For the implementation of OpenSSL, there was a measurable time difference in MAC verification. In case a message was correctly padded, the MAC was checked as well; however, in case the padding was not correct, the verification of the MAC is skipped. This gives the attacker space to learn more about the transmitted message.

Heartbleed
The Heartbeat extension is an extension for the TLS protocol that allows the communicating parties to send a Heartbeat Request message; the Heartbeat extension is specified in RFC 6520 [29]. This message contains a payload, typically a text string, along with the payload’s length. The receiving party must send the same payload back to the sender to acknowledge that the connection is still active.
OpenSSL is a widely used library that can be used to implement the TLS protocol; this library supports the Heartbeat extension. However, some versions of the OpenSSL library had an implementation issue; this caused the so-called Heartbleed vulnerability. In the affected versions of OpenSSL, memory buffers were allocated based on the sent payload length, without regard to the actual size of that message's payload. By setting a payload length greater than the actual payload, the server sends a message back containing the payload and whatever happens to be in its active memory. This can lead to the disclosure of confidential information [30].

Heartbleed was introduced in December 2011 and publicly disclosed in April 2014. The Heartbleed vulnerability was classified as a buffer over-read which is a situation where more data can be read than should be allowed.
3 Digital certificates

This chapter explains digital certificates in more detail. Digital certificates are used to authenticate a communicating party in the TLS protocol. Section 3.1 gives a brief introduction about what digital certificates are. In order to bind an agent's identity and public key to a digital certificate, the certificate is signed by a trusted third party. This third party is called a Certificate Authority; section 3.2 describes the role of this third party in more detail. For the TLS protocol, the X.509 certificate standard is a commonly used standard; section 3.3 goes in more detail about this standard. Furthermore, to achieve authenticity in the TLS protocol, it is important that the certificate is validated. Section 3.4 describes certificate validation methods in more detail.

3.1 Introduction

Authentication is an important feature of the TLS protocol; this can be achieved with digital public key certificates\(^2\). A certificate binds an agent and its public key, by cryptographically signing this information by a trusted entity. In the TLS Handshake protocol, the information on the certificate can be used to authenticate the communicating party, and the public key can be used to set up an encrypted channel. An important part of the authentication process is to validate the signature of the provided certificate; this process will be discussed in more detail in section 3.4.

The management and standards of the certificates are not specified in the protocol specifications of the SSL or TLS protocol. This is addressed outside the scope of these protocols [31]. According to RFC 2828 [32], the term certificate refers to “that binds a system entity’s identity to a public key value, and possibly to additional data items; a digitally-signed data structure that attests to the ownership of a public key”.

In order to satisfy the statement above, a certificate comprises at least the following three pieces of information:

- **Public key**: Cryptographic key of an entity.
- **Naming information**: Identification of the owner of the certificate, the most prominent is the domain name.
- **Digital signature(s)**: Evidence that the attributes of the certificate belong together.

Additionally, certificates may also contain additional information, such as the start and expiration date and many other properties [8].

3.2 Certificate authority

As described in section 3.1, certificates are digitally signed as evidence that the information on the certificates belongs together. When the signature is performed by a trusted entity, this signature can also be considered as proof of the validness of the information itself on the certificate.

Such a trusted party is called a Certificate authority (CA) and is responsible for issuing and revoking certificates for agents. A CA is an organization that processes the request of its customers which needs a digitally signed certificate. Based on this request, the CA verifies the information and potentially issues a signed certificate; the signature on this certificate is performed with the private key of the CA itself [8]. According to RFC 2828 [32], the Public Key

---

\(^2\) The term “certificate” will be used to refer to a digital public key certificate for the rest of this thesis
Digital certificates

Infrastructue (PKI) is defined as: "A system of CAs that perform some set of certificate management, archive management, key management, and token management functions for a community of users in an application of asymmetric cryptography".

When a communicating party wants to verify a certificate of another party, it needs a set of trusted CAs. Usually, those trusted entities’ certificates are shipped in the operating system or software. When the received certificate is signed by either of those CAs, the information of the certificate can be considered as valid. Since the validness of the certificate can be assured, which is based on trust, authenticity can be achieved [8].

3.2.1 Chain of trust
Certificates can be signed hierarchically; meaning that certificates can be signed by a trusted party and that those parties use their certificate, to sign other certificates. This forms a so-called chain of trust. In terms of CAs, we can make the following distinguishes [31]:

- Root CA: Those kinds of CAs are to be trusted by a client. Meaning that the client has a preconfigured set of trusted root CAs. The certificates of the root CAs are self-signed, this does not provide any guarantee but is avoidable since the hierarchy is finite and needs a top-level.
- Intermediate CA: Those kinds of CAs do not need to be directly trusted by the client. Instead, when its root CA is considered as trusted, the intermediate CA can be considered as trusted as well.

![Diagram of chain of trust](image)

Figure 4: Chain of trust

Equipped with one or several certificates of root CAs, the communicating party can try to find the certification path from a root certificate to a leaf certificate. Furthermore, the communicating party can also trust an intermediate CA, in that case not the whole sequence of certificates needs to be verified. The user only needs to trust one or more anchors in the certificate sequence, this hierarchy is called the chain of trust [8]. Figure 4 illustrates an example of such a chain.

3.2.2 Baselines
In order to be considered as a CA, the organization needs to be trusted by a community of users [31]. The baselines to be considered as a CA can be decided by companies such as CAB Forum [33] or PKIoverheid [34]. These baselines give requirements for CAs which need to be satisfied to be considered as trusted by their community. Hence, the trustworthiness of a CA depends on whether it meets certain baselines and its reputation.
The baselines which are satisfied by a CA can also determine the level of security for a TLS connection. To request a certificate at a CA, the entity which is requesting the certificate has to satisfy certain requirements as well. There are three important requirements which also result in different kind of certificates [35]:

- **Domain validation**: CA validates that the entity owns the domain. This is usually done by that the CA sends an email to a domain owner or that the entity has to host a certain file or token on the domain which is validated by the CA. Because of the minimal checks performed, domain validated (DV) certificates are mostly issued quicker than other types.

- **Organization validation**: CA validates that the entity owns the domain and organization. The CA has to do the same check as with DV certificates. On top of that, it also needs to confirm that the entity owns the stated organization by going through public databases. Since there are additional checks in place, the issue process of organization validated (OV) certificates tend to take longer than DV certificates.

- **Extended validation**: CA validates that the entity owns the domain, organization and the legal entity that controls the domain. The CA has to do the same check as with OV certificates. On top of that, Extended validated (EV) certificates are only issued once an entity passes a strict authentication procedure. Furthermore, those kinds of certificates can only be issued by a subset of CAs. EV certificates tend to have the longest issue process.

Figure 5 illustrates Northwave’s certificate, this certificate is issued by the CA called Let’s Encrypt [36]. Let’s Encrypt is an automated CA which issues DV certificates. An entity can request a certificate by providing its public key and domain name; once this request is processed, the entity has to perform a challenge to prove that it owns the domain [37]. This challenge will be automatically validated according to the Automatic Certificate Management Environment (ACME) standard which is defined in RFC 8555 [38]. Once the entity completes the challenge, its certificate will be issued. Let’s Encrypt gives the entity the choice to do either of the following challenges [39]:

- **HTTP-01 challenge**: Let’s Encrypt sends a token to the entity’s ACME client. Using this token, the entity uploads a file named after the token in a specific accessible directory on its web server; for example: `http://<DOMAIN>/.well-known/acme-challenge/<TOKEN>`. Once this file is ready, the entity’s ACME client tells Let’s Encrypt that the file is ready so that it can automatically start retrieving it. This challenge proves that the entity has access to its webserver and therefore owns the domain.

- **DNS-01 challenge**: Let’s Encrypt sends a token to the entity’s ACME client. Using this token, the entity sets a DNS TXT record on its DNS server for the challenge [40]; for example: `_acme-challenge.<DOMAIN>`. This TXT record contains information such as the token. Once the DNS TXT record is set, the entity’s ACME client tells Let’s Encrypt that it is ready so that it can automatically start retrieving it. This challenge proves that the entity has access to its DNS server and therefore owns the domain.

In both challenges described above, Let’s encrypt sends a token to the entity. In this message, Let’s Encrypt also sends a nonce. When the entity completes the challenge, it also has to include this nonce, signed with its public key, in the request. The entity’s public key was sent as part of the entity’s certificate request. By signing the nonce, the entity proves that it owns its public key. This approach prevents replay and man-in-the-middle attacks [37]. Furthermore, Let’s Encrypt only issues DV certificates. This means that when a company is using this CA for its website, it only proved that it is in possession of the domain. Therefore, the user who is visiting that website only has the guarantee that the website is owned by that entity. It does not guarantee that the company indicated is a real company.
Digital certificates

3.3 X.509 standard
The X.509 standard is a commonly used certificate standard for the TLS protocol. This standard has three versions which are released over the years; X.509 version 3 is the most recent. Over the years, fields have been added to the certificate standard [31]. According to IBM, version 1 is therefore widely deployed since it is the most generic variant [41]. Figure 5 illustrates Northwave’s X.509 certificate, this certificate is used for Northwave’s website.

3.3.1 Certificate format
The format of the X.509 certificate is specified in Abstract Syntax Notation One (ASN.1) [42] and is typically encoded with: Basis Encoding Rules (BER), Distinguished Encoding Rules (DER) or Packet Encoding Rules (PER) [31]. The encoded result produces a series of bytes which are suitable for transmission. The X.509 standard consists of the following fields:

- **Version**: Specifies the X.509 version in use.
- **Serial number**: Unique integer identified by the certificate issuer.
- **Algorithm ID**: Object Identifier of the algorithm which is used to sign the certificate.
- **Issuer**: Domain name of the issuer.
- **Validity**: Validity period, defined by start and finish date, of the certificate.
- **Subject**: Domain name of the owner (subject) of the certificate.
- **Subject Public Key Info**: Certified public key together with the algorithm.
- **Authority Key Identifier**: Optional information related to the issuer, only available in X.509 version 3.
- **Subject Key Identifier**: Optional information related to the owner of the certificate, only available in X.509 version 3.
- **Extensions**: Optional extensions, only available in X.509 version 3.

Figure 5: Northwave’s digital certificate
3.3.2 Additional extensions and techniques
Below follows a brief description of several additional extensions and techniques which are supported by the X.509 certificate standard.

3.3.2.1 Basic constraints
In order to get a better view of the hierarchy of the chain of trust, the Basic constraints field can be used. The Basic constraints is an extension field of the X.509 version 3 standard. This extension field comprises the fields CA and pathLenConstraint. The field CA is a boolean which identifies whether the certificate is a CA certificate. Hence, a certificate should only sign other certificates when CA is true. This field is mainly used when a CA issues a certificate to an intermediate CA, which allows the intermediate CA to sign other certificates. The field pathLenConstraint identifies the maximum depth of valid certification paths to other (intermediate) CA certificates [43]. For instance, the root CA’s certificate in figure 4 has a value of 1; the intermediate CA’s certificate pathLenConstraint has a value of 0.

3.3.2.2 Key usage
The key on the certificate can have a specific purpose. Therefore, the Key usage field, which is an extension field of the X.509 version 3 standard, can be used to define the purpose of this key. CAs can include this extension on its certificates to identify that its key can be used to validate digital signatures of its issued certificates [43]. Table 2 illustrates the possible values for this field, multiple values in this field are possible.

<table>
<thead>
<tr>
<th>Key usage</th>
<th>Value</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td>digitalSignature</td>
<td>0</td>
<td>Key can be used to verify signatures, other than signatures on certificates and CRLs. This value is set when the key can be used in an entity authentication service, a data origin authentication service, and/or integrity service.</td>
</tr>
<tr>
<td>nonRepudiation</td>
<td>1</td>
<td>Key can be used to verify signatures, other than signatures on certificates and CRLs. This value is set when the key can be used to provide repudiation service which protects against the signing entity falsely denying some action.</td>
</tr>
<tr>
<td>keyEncipherment</td>
<td>2</td>
<td>Key can be used to enciphering private or secret keys, for example for key transport.</td>
</tr>
<tr>
<td>dataEncipherment</td>
<td>3</td>
<td>Key can be used to directly enciphering raw user data without the use of intermediate symmetric key establishment.</td>
</tr>
<tr>
<td>keyAgreement</td>
<td>4</td>
<td>Key can be used for key agreement such as Diffie-Hellman.</td>
</tr>
<tr>
<td>keyCertSign</td>
<td>5</td>
<td>Key can be used to verify signatures of other certificates. This value should only be set when the Basic constraints’ CA field is also set.</td>
</tr>
<tr>
<td>cRLSign</td>
<td>6</td>
<td>Key can be used to verify signatures on certificate revocation lists.</td>
</tr>
<tr>
<td>encipherOnly</td>
<td>7</td>
<td>Used only when keyAgreement is set, by also setting this value, the key can only be used to encipher data while performing a key agreement.</td>
</tr>
<tr>
<td>decipherOnly</td>
<td>8</td>
<td>Used only when keyAgreement is set, by also setting this value, the key can only be used to decipher data while performing a key agreement.</td>
</tr>
</tbody>
</table>

Table 2: X.509 extension Key usage values [43]
3.3.2.3 Certificate Revocation List

As mentioned above, a certificate has a validity period. However, in some cases, a certificate needs to be revoked before its expiration date. This is needed when for instance a private key of an entity in the chain of trust is compromised, or when a CA goes out of business. The revocation of certificates can be done with a Certificate Revocation List (CRL), which is specified in RFC5280 [43]. The CRL is a list, periodically published and signed by the CA, which contains all the certificates which are revoked by the CA before its expiration date. Therefore, the CRL can be a very large list [8].

3.3.2.4 CRL Distribution Points

In order to specify how the CRL can be obtained, the CRL Distribution Points extension field can be used. This extension supports X.509 version 2 and above and is specified in RFC5280 [43]. The CRL Distribution Points extension field is a sequence of DistributionPoint which consists of three fields, each of which is optional: distributionPoint, reasons, and cRLIssuer. To have a valid DistributionPoint record, either reasons or cRLIssuer needs to be present. Furthermore, if the certificate issuer is not the issuer of the CRL, the cRLIssuer field must be present.

The distributionPoint field comprises either a sequence of general names (URI) or a single value. When it contains multiple names, each name describes a different mechanism to obtain the CRL. The mechanism can be a directory name that can be obtained by the protocol which the application is using; the actual implementation for this is a local matter and out of the scope of the RFC. The field reasons is a bit string field which contains the reasons why the certificates in the CRL list are revoked. The cRLIssuer field contains the name of the issuer of the CRL list.

3.3.2.5 OCSP

With the approach of CRL, the requester receives a list of all the revoked certificates by the CA. Since this list can be very large, the trend goes to retrieving the online status information of a certificate directly. This can be achieved with the Online Certificate Status Protocol (OCSP); the OCSP is specified in RFC6960 [44]. By using OCSP, the client can request the revocation status of a specific certificate, this puts less burden on the network compared to the CRL approach.

The requester sends a request with the following payload: protocol version, service request, target certificate identifier and optional extensions. The response is digitally signed and consists of the following: version response syntax, identifier of responder, time of response, response for each certificate in request, optional extensions, signature algorithm OID and the signature which is computed across a hash of the response. The response of each certificate consists of target certificate identifier, status value, response validity interval and optional extensions. The possible status values for the certificate are good, revoked or unknown.

3.3.2.6 Certificate Transparency

Since there are many certificates issued daily, it is hard to keep track of the validity of these operations. Therefore, the Certificate Transparency (CT) protocol is proposed in RFC 6962 [45]. The CT protocol focuses on publicly monitoring and auditing issued certificates. This allows a community to efficiently identify mistaken or maliciously issued certificates. Anyone
can submit certificates to the logs for public auditing. However, the goal is that the CAs will make use of this protocol. The log is a single, ever-growing, append-only tree that is public and can, therefore, be audited by anyone of the internet. This allows, for instance when an attacker issues a certificate for someone’s domain, that the domain owner has a transparent view over the certificates which are issued for its domain.

### 3.4 Certificate validation

An integral process of the TLS protocol is authentication, this can be achieved by transmitting certificates with the TLS Handshake protocol in a Certificate message. This message contains all the certificates in the chain which can be used to validate the entity’s certificate. However, building and validating the path needs to be performed by the receiving party of the certificate. This section discusses different methods to achieve certificate validation.

#### 3.4.1 Certification path processing

The goal for the party which receives a certificate is to validate the certificate chain until a trust anchor is reached. This process is called certification path processing and consists of two phases [46]:

1. **Path construction:** Discovery of one or more candidate certification paths from the provided certificate to a trusted anchor.
2. **Path validation:** Validating that each certificate in the path is within the validity period, has not been revoked and has a valid signature.

The certification path process has not been standardized and needs to be implemented in the application logic. However, there are implementation guidelines for certification path validation defined in RFC5280 [43]. The actual implementation of the validation can be either implemented locally in the software, remotely delegated to a trusted third party, or a combination of both. Figure 6 illustrates the certification path of Northwave’s certificate, this certificate is used for Northwave’s website.

![Figure 6: Northwave’s digital certificate’s certification path](image-url)
3.4.1.1 Path construction
Certification paths can be either constructed in a forward direction (from end-entity certificate to trust anchor certificate) or a reverse direction (from trust anchor certificate to end-entity certificate) [47]. The simplest hierarchy is a tree with a single root CA, in that case, only one certification path exists. However, in practice, more structures are possible, using multiple root CAs, intermediate CAs, and CAs that issue cross-certificates. In such a structure, a certification path may not be unique, and multiple certification paths may exist. Because of the complexity of this problem, this has been a point of discussion in the past few years [31].

The actual implementation depends on what algorithm is implemented on the receiver’s side. An important process of such an algorithm is chaining the certificates. There are several ways to do that; those are described in more detail below. Note that there may be many other ways to chain certificates other than the ones described [46].

Name chaining is the most basic level of chaining certificates, based on the Subject and Issuer field. This means that the Subject in the root certificate has to match the Issuer in the next certificate in the path, and so on. However, this approach is not robust since that CAs can have multiple signing pairs, which means that the CA’s name can correspond with multiple certificates. By name chaining, those different certificates cannot be distinguished from each other.

Chaining certificates can also be accomplished with a key identifier. As key, the Authority Key Identifier (AKID) and Subject Key Identifier (SKID) fields, which are certificate extensions of the X.509 version 3 standard, can be used to chain certificates. In this type of chaining, the SKID in the root certificate has to match the AKID in the next certificate in the path, and so on. AKIDs are used to distinguish one public key from another when a CA has multiple keys; and, SKIDs are linked to a specific public key of an agent.

3.4.1.2 Path validation
Once the path is constructed, the certificates in the certification path needs to be validated. For each certificate, their properties need to be validated, such as matching properties in the certificate chain and verifying the signature on each certificate. If the criteria of all the certificates in the chain are met, the chain of trust is established, and the provided certificate can be accepted.

3.4.2 HTTP Public Key Pinning
HTTP Public Key Pinning (HPKP) is a security mechanism, specified in RFC 7469 [48], that pins a public key for a host on the first encounter for a certain amount of time. When the client sends a request to a webserver for the first time, the web server can send a set of hashes of public keys. Those public keys must appear in the chain of trust for future connections to the same domain. The web server communicates this HPKP policy to the user via an HTTP response header in the field named Public-Key-Pins and/or Public-Key-Pins-Report-Only.

When the Public-Key-Pins field is set, the user should strictly enforce the provided public key hashes and report it when it is not matching. The field Public-Key-Pins-Report-Only is more lenient when this field is set, the user does not have to strictly enforce the provided public key hashes but report when the hashes are not matching. Both fields contain the following parameters [48]:
- pin-sha256: Acceptable hash of the public key, there can be multiple parameters of this type.
Digital certificates

- includeSubDomains: Set when this pin also applies to other subdomains of the host, this parameter is optional.
- max-age: The timeframe in which the public key(s) can be pinned.
- report-uri: Uri which the report needs to be sent to when the hashes do not match.

However, the implementation of HPKP is complex and there is also a chance of possible or accidental misuse of this mechanism. Therefore, browsers deprecated and removed the support for HPKP in 2018 [49, 50]. Moreover, HPKP tends to be used by browsers; HPKP was not supported by the Android framework [51]. Instead, Android supports certificate pinning which is discussed in section 3.4.3.

3.4.3 Certificate pinning
Certificate pinning is a certificate validation technique that associates a host with a specific certificate in the chain of trust. Since certificate pinning is a validation technique for certificates, it needs to be implemented as part of the TLS Handshake protocol. The implementation of the certificate pinning technique is part of the application logic and therefore it needs to be implemented in the application. Certificate pinning can prevent man-in-the-middle attacks for the TLS protocol, with this technique it is harder for an attacker to inject its own certificate during the TLS handshake.

When more than one certificate is accepted for a host, the receiver can hold an acceptance list, which is called a pinset, for that host. There are two approaches regarding to when a certificate can be pinned [52]:
- Development time: Hardcoded in the software, this is the most secure approach since preloading of the certificate out of band usually means the attacker cannot taint the pinned certificate.
- First encounter: Certificate is pinned on the first encounter; this is less secure. Because, when the first connection was compromised by a man-in-the-middle attacker, the attacker’s certificate can be pinned. Note that this disadvantage also applies to HPKP which is discussed in section 3.4.2.

A downside of certificate pinning is that the application requires an update when the pinned certificate in the chain is updated. There are three different approaches regarding on what level in the certificate chain, a certificate can be pinned for a host [53]:
- Pinning leaf (end) certificate: Pins entity’s certificate which also reduces the attack surface to a minimum. But also, the least flexible.
- Pinning intermediate certificate: Pins the intermediate certificate of entity’s certificate in the chain, this approach has a larger attack surface than the previous but is more flexible.
- Pinning root certificate: Pins the root (CA) certificate of entity’s certificate in the chain, this approach has the largest attack surface but is also the most flexible.

The least flexible approach is pinning the leaf certificate, this is the communicating party’s certificate. This is the least flexible approach, when a new certificate is enrolled for a host, the pinned certificate needs to be updated in the application logic as well. However, this is the most secure approach compared to for instance pinning the intermediate certificate. When the developer chooses the pin the intermediate certificate, the end-entity can update its leaf certificate without updating the application when the same intermediate CA is used. However, the attack surface will be bigger; when an attacker can somehow get a certificate signed by the same intermediate CA, this certificate could be used to perform a man-in-the-middle attack.
As described above, any certificate in the chain can be targeted to be pinned. Therefore, when the leaf certificate is not pinned, the certification chain needs to be calculated. This process is described in more detail in section 3.4.1. Furthermore, this also means that the developer can choose to use a self-signed certificate for its server. By pinning this leaf certificate, the chain of trust does not have to be validated and also results in that it does not rely on the trusted CAs which are shipped with the operating system of the user [53].
4 Certificate pinning implementation

This chapter introduces certificate pinning, which code examples, in an Android environment. Section 4.1 starts with a brief introduction about the Android operating system. This gives a better understanding of certain behavior which is useful for subsequent chapters. Since this paper is about the TLS protocol in an Android environment, section 4.2 describes how the TLS protocol can be implemented. Furthermore, the main focus of this thesis is to break the TLS protocol in an Android environment with the certificate pinning technique implemented. Therefore, section 4.3 gives a detailed overview, with code examples and an alternative library, about how certificate pinning can be implemented in an Android environment. To get a broader view, section 4.4 briefly describes how the TLS protocol can be implemented in an iOS environment; it will also briefly explain why iOS is out of the scope for this project.

4.1 Introduction

This section gives a brief introduction of the Android environment. This information gives a better understanding of the techniques being used; this is necessary to understand further chapters in this thesis.

The Android operating system is a Linux-based software stack created for a wide array of devices. The base of the Android operating system is the Linux kernel, this allows Android to use key security features of it and allows device manufacturers to develop hardware drivers for this well-known kernel [54]. Each Android application is executed in its own security sandbox, this introduces the following security features [55]:

- The Android operating system is a multi-user Linux system in which each Android application is a different user.
- When the Android application is executed, it will get a Linux user ID assigned by the operating system. The system sets permissions for all the required resources for the Android application’s user profile only. Because of this, the application can only access resources that are required for execution.
- Each Android application is executed in its own virtual machine (VM), so an Android application runs in isolation from other applications.

Since the Android operating system is Linux-based, it also facilitates a Unix shell. This shell can be accessed with the Android Debug Bridge (adb). Adb is a client-server command-line tool that enables desktops to communicate with Android devices. It supports a variety of device actions, such as installing and debugging applications, and it provides access to the Unix shell that can be used to run a variety of commands; this makes the Android environment very similar to the Linux environment. Adb is a client-server program which means that the adb runs on the device and it also needs to be installed on a desktop. By running adb on the desktop and connecting the device via a USB cable, adb can be used [56].

The Android Software Development Kit (SDK) can compile the source code of an Android application, along with any data and resource files, into an Android package (APK); this file type is an archive file with an .apk suffix. This APK file contains all the contents of an Android application and can be used to install the application on an Android device [55]. The Android operating system requires that the APK file is digitally signed with a certificate. Without this signature, the Android application cannot be installed or updated. By signing the APK, the signing tool attaches the author’s certificate to the Android application. By doing this, the Android application is associated with the author. Furthermore, this helps the Android operating system to ensure that any future updates of the Android application are authentic and released by the original author. Note that the private key, also called the app signing key,
which is used to sign the Android application must be kept secret; the corresponding certificate is not confidential [57].

4.2 Android TLS implementation

The Android operating system is shipped with a set of trusted CA root certificates which can be used to authenticate a server for a TLS connection. As of Android version 4.2 (Jelly Bean), Android contains over 100 well-known trusted CAs which are updated in each release [58]; this update might also include blacklisting CAs, this happens when a CA is compromised or goes out of business. The trusted CAs are stored in the device’s Keystore [59].

Users can also manually add CA certificates to the device’s Keystore. For Android versions 6.0 and lower, those user-added certificates are automatically considered as trusted. However, for Android versions above 6.0, user-added certificates are by default not considered as trusted anymore. The reason for this change is that the Keystore system enforces the safe-by-default principle; it reduces the application’s attack surface [60]. However, the user-added CA certificates can be considered as trusted for an Android application, when the developer enables a setting in the Network configuration file of the application [61].

In order to set up a TLS connection in an Android environment, Java’s default java.net library can be used. This library uses the device’s Keystore to validate the server’s certificate chain. Figure 7 illustrates an example of a request using the URLConnection class, this class is part of the java.net library. When the function openConnection() is called, a TLS handshake will be performed; this also includes the validation of the server’s certificate. After the handshake is completed, a TLS session is established. Once the session is established, the actual request will be performed, and its result will be returned as part of this function. Note that for this example it is assumed that the server’s CA certificate is in the device’s Keystore.

```java
URL url = new URL("https://cees.pwlab.nl/");
URLConnection urlConnection = url.openConnection();
InputStream in = urlConnection.getInputStream();
copyInputStreamToOutputStream(in, System.out);
```

Figure 7: Example TLS connection implementation [58]

The URLConnection class is the superclass of the HTTPURLConnection and HTTPSURLConnection classes [62]. By calling the openConnection() method, an instance of either of those subclasses is returned, depending on whether the HTTP or HTTPS (HTTP over TLS) protocol is used for the request. The two subclasses are using Android’s default instance of the KeyStore, TrustManager and SSLSocket classes. Those instances are used to perform the TLS handshake, including the authentication of the server using the device’s Keystore [63]; those classes are discussed in more detail in section 4.3.
4.3 Android certificate pinning implementation
As described in section 4.2, the HTTPSURLConnection class can be used to establish a connection over HTTPS between the Android application and the server. By using this class, Android’s default settings are used to establish a TLS session to process the request to the server.

However, the HttpsURLConnection class can also be used to implement certificate pinning. This is Android’s default approach and uses functions that are embedded in the Android operating system. Certificate pinning can be achieved by manually configuring a Keystore with the pinned certificate, and use this Keystore only, to establish TLS session(s). By doing this, Android’s default Keystore will not be used to authenticate the server, but the manually created Keystore instance instead. To get a better understanding of this process below follows a brief explanation about the classes involved in this process:

- CertificateFactory: Factory to create an instance of a Certificate, Certpath or CRL.
- Certificate: Facilitates the storage of the certificate information which is applicable for different certificate formats.
- KeyStore: Secure storage facility for cryptographic keys and certificates. The material which is stored in this container can be used for cryptographic operations while the material itself is non-exportable.
- TrustManagerFactory: Factory to create an instance of a TrustManager.
- TrustManager: Handles trust material for managing trust decisions. This interface is focused on making decisions for a given KeyStore.
- SSLSocketFactory: Factory to create an instance of a SSLSocket.
- SSLSocket: Facilitates a secure socket for the SSL and TLS protocols. This class adds the following layer of security protections over the underlying network transport protocol: integrity protection, authentication and confidentiality. The protection itself is specified by the cipher suite which is negotiated during the handshake phase which is also performed with this class.
- SSLContext: Represents a secure socket protocol implementation that acts as a factory for secure socket factories. This class needs to be initialized with an optional set of key managers, trust managers and a source of secure random bytes.

Figure 8 illustrates a code snippet about an implementation of certificate pinning using Android’s default library. This implementation is using the classes which are described above. Android’s default library pins the certificate as a whole, in this example this certificate is located in the local directory of the application. On line 3, follows the function call to load the certificate in as an InputStream instance from the local directory. To cast the certificate’s InputStream to a Certificate instance, the CertificateFactory is being used on line 6. This Certificate instance is then added to a local Keystore instance on line 15. Once the Keystore with the pinned certificate is initialized, this instance is added to a TrustmanagerFactory instance on line 20. Finally, on line 24, a SSLContext instance is initialized with the TrustManager instance which is created using the TrustmanagerFactory instance. The SSLContext instance contains the pinned certificate and can be used for future requests. This can be achieved by using the SSLContext to generate a SSLSocketFactory as illustrated on line 29.

---

3 The term “Android’s default library” will be used to refer to this approach for certificate pinning for the rest of this thesis
Certificate pinning implementation

```java
CertificateFactory cf = CertificateFactory.getInstance("X.509");
// Load server's certificate from a file
InputStream caInput = new BufferedInputStream(new FileInputStream("my-certificate.crt"));
Certificate ca;
try {
    ca = cf.generateCertificate(caInput);
} finally {
    caInput.close();
}

// Create a KeyStore containing our certificate
String keyStoreType = KeyStore.getDefaultType();
KeyStore keyStore = KeyStore.getInstance(keyStoreType);
keyStore.load(null, null);
keyStore.setCertificateEntry("ca", ca);

// Create a TrustManager that trusts the certificate of our KeyStore
String tmfAlgorithm = TrustManagerFactory.getDefaultAlgorithm();
TrustManagerFactory tmf = TrustManagerFactory.getInstance(tmfAlgorithm);
Mf.init(keyStore);

// Create an SSLContext that uses our TrustManager
SSLContext context = SSLContext.getInstance("TLS");
context.init(null, tmf.getTrustManagers(), null);

// Tell the URLConnection to use a SocketFactory from our SSLContext
URL url = new URL("https://oesg.pnlab.ah/");
HttpsURLConnection urlConnection = (HttpsURLConnection) url.openConnection();
urlConnection.setSSLSocketFactory(context.getSocketFactory());
InputStream in = urlConnection.getInputStream();
copyInputStreamToOutputStream(in, System.out);
```

Figure 8: Example certificate pinning implementation with Android's default library

Another way to implement certificate pinning is by using the OKHttp library. OkHttp is an API which initializes an HTTP client. This client supports modern TLS features such as certificate pinning; OkHttp is compatible with Android versions 5.0 and above. Furthermore, it uses the platform's built-in TLS implementation to set up a TLS connection.

Figure 9 illustrates a code snippet which is an implementation of certificate pinning using the OKHttp library. In comparison with Android's default library, illustrated in figure 8, the OKHttp library pins a certificate by a hard-coded, as a string, fingerprint of the certificate; Android's default library does this by pinning the certificate as a whole. On line 5, follows the function call to pin a certificate for a specific host. This results in a CertificatePinner instance which is added to the OKHttpClient; this client instance is built on line 7. The OKHttpClient instance contains the pinned certificate and can be used for future requests. This can be achieved by using this OKHttpClient instance, as illustrated on line 15, to initiate a request. OKHttp's implementation of certificate pinning is simplified compared to Android's default library.
Certificate pinning implementation

Figure 9: Example certificate pinning implementation with the OkHttp library [72]

Note that there may be various other libraries to implement certificate pinning in an Android environment. According to Northwave, Android’s default library and the OKHttp library are the most commonly used libraries by their customers to implement certificate pinning.

4.4 iOS implementation

The iOS operating system has a networking feature called App Transport Security (ATS). This feature is embedded in the iOS applications and, improves privacy and data integrity for its connections. ATS requires that all HTTP connections which are made with Apple’s default library are using HTTPS; HTTP connection attempts fail [73]. Furthermore, it also requires that the connection is using the TLS protocol versions 1.2 or above using a cipher that supports forward secrecy [74]. In the application’s production environment, ATS is enabled by default when it is targeting iOS versions 9.0 or higher; however, it can manually be disabled [74]. At WWDC 2016, Apple announced that applications which are submitted to the App Store will be required to use ATS by the end of that year. However, Apple has extended that deadline, the current deadline is unknown [75].

The NSURLSession class, embedded in the iOS SDK, coordinates a group of related network data transfer tasks by using one or more sessions. This class supports the data, file, FTP, HTTP and HTTPS URL schemes. Also, ATS is by default enabled for the connections made with this class. The NSURLSession can be instantiated with one of the three kinds of configurations [76]:

- Default session: Allows the client to obtain data incrementally.
- Ephemeral session: Similar to the default session, but doesn’t write caches, cookies, or credentials on the device.
- Background session: Perform uploads and downloads of content in the background while the application is not actively running.
As in Android, Apple devices have a Trust Store that contains trusted CA root certificates that are preinstalled in the iOS operating system. This Trust Store is used to authenticate the communicating party for a TLS connection. The set of trusted certificates depends on which version of iOS is running on the device. The iOS Trust Store contains three categories of certificates [77]:

- Trusted certificates to establish a chain of trust.
- Untrusted certificates which are not blocked. When this kind of certificate is used, the user will be prompted to choose whether to trust it.
- Blocked certificates which are considered to be compromised and will not be accepted to establish a TLS session.

Apple also offers a service to detect and flag different Apple devices. This service is called the DeviceCheck which is a set of APIs. DeviceCheck can be implemented into an iOS application to identify a device; an example is, to identify whether a device has already taken advantage of a promotional offer that is provided by the application. When the developer is using this API, it can also flag devices that are determined to be fraudulent. Furthermore, the DeviceCheck API can also verify that the application backend is communicating with an authentic Apple device; however, detecting jailbroken devices is out of the scope of this API. Unfortunately, Apple does not specify how the checks are performed [78].

The DeviceCheck API is compatible with the iOS operating system only. Furthermore, the SafetyNet Attestation service, discussed in more detail in chapter 5, is only compatible with the Android operating system. Since the SafetyNet Attestation service only works on the Android operating system, the iOS operating system is out of the scope for this thesis.
5 Android SafetyNet service

This chapter describes the Android SafetyNet service. This service can be used to make anti-abuse decisions. Section 5.1 gives an introduction about this service by describing the different API functionalities. Furthermore, this paper focusses on the SafetyNet Attestation service, which is part of the overall Android SafetyNet service, therefore section 5.2 gives an in-depth analysis of this service. Section 5.3 describes a brief explanation, with a code example, of how the SafetyNet Attestation service can be implemented into an Android application.

5.1 Introduction

The Android SafetyNet is a set of services and APIs that helps to protect an Android application against security threats such as device tampering, bad URLs, potentially harmful apps, and fake users [79]. The Android SafetyNet service is split in the following APIs:

- SafetyNet Attestation: Anti-abuse service that allows application developers to assess the Android device their Android application is running on. Also, with this service, it can be determined whether the backend server is interacting with the genuine Android application [6].
- SafetyNet Safe Browsing: Service that can determine whether a URL has been marked as a known threat by Google [80].
- SafetyNet reCAPTCHA: Service that can suspect whether a user who is interacting with the Android application, might be a bot instead of a human. When this is suspected, this API serves a CAPTCHA that needs to be solved before continuing [81].
- SafetyNet Verify apps: Service that can determine whether the Android device where the Android application is running on, has other potential applications installed. The goal of this service is to make the Android ecosystem as safe as possible, this results in protecting the security and privacy of the user [82].

The Android SafetyNet libraries are shipped as part of the Android operating system. Therefore, it does not require to download an additional library to use it. However, to use those libraries effectively, it needs to be implemented in the Android application and its backend server.

According to Northwave, the SafetyNet Attestation service is one of the reasons why the Android-SSL-Trustkiller tool is not working anymore [5]. This tool was used by Northwave to bypass certificate pinning; The Android-SSL-Trustkiller is discussed in more detail in section 7.1. Since the SafetyNet Attestation service patched the previous solution to bypass certificate pinning, section 5.2 describes a detailed analysis of this SafetyNet Attestation’s validation process.

5.2 SafetyNet Attestation process

The SafetyNet Attestation service is an anti-abuse service that allows, when implemented in the Android application and its server, to assess the Android device that the Android application is running on. Furthermore, this service can determine whether the application’s backend server is interacting with the corresponding genuine Android application on a genuine Android device. The SafetyNet Attestation service can assess the validness of the following [6]:

- Application’s package name.
- Certificates that are used to sign the application’s installation file.
- Android device’s integrity by determining whether it is likely that the device is tampered with.
As part of Android’s device integrity check, the SafetyNet Attestation service can detect whether an Android device is rooted; rooted Android devices are discussed in more detail in section 8.2. However, according to SafetyNet’s documentation, the SafetyNet Attestation service is not designed to purely check whether devices are rooted. Instead, the service is designed to check the overall integrity of Android devices [6].

5.2.1 Workflow overview
In order to validate the content described above, the SafetyNet Attestation service has a specific workflow. For this workflow, there needs to be an implementation in the Android application and its backend server. Furthermore, in this workflow, Google’s server is used as a third party to add an integrity check on the provided information. Hence, the Android device where the Android application is running on needs to be able to make a connection to Google’s server [6].

In order to use the SafetyNet Attestation service, an API key needs to be requested, at Google’s website, for the Android application. This API key is used to initiate the SafetyNet Attestation check by calling SafetyNet’s API. Based on this API key, there is a default quota allotment of 10,000 requests per day to the Google server. However, upon request, a higher quota can be requested [83]. The SafetyNet Attestation check is not performed regularly while the Android application is running; but rather a check which is performed during the startup of the Android application. The SafetyNet Attestation service uses the following workflow [6]; this step-by-step process is also illustrated in Figure 10:

1. Application server sends a nonce to the Android application.
2. Android application calls the SafetyNet Attestation API. This call includes the provided nonce.
3. SafetyNet Attestation service evaluates the runtime environment and requests a signed attestation of the assessment results from Google’s server.
4. Google’s server sends the signed attestation to the SafetyNet attestation service on the device.
5. SafetyNet Attestation API forwards this signed attestation to the Android application.
6. Android application forwards this signed attestation to the application server.
7. Application server validates the signature of the provided signed attestation by using Google’s certificate(s). The server also evaluates the payload of the attestation, this also includes a check whether the nonce in this payload matches with the nonce which was initially sent (step 1). Furthermore, the application server makes anti-abuse decisions, based on evaluation on the rest of the payload, and communicates the findings back to the Android application.
5.2.2 Workflow steps
This section gives an in-depth description of the steps in the workflow of the SafetyNet Attestation check [6]; those steps are briefly discussed in section 5.2.1 and is illustrated in figure 10.

1. **Server sends nonce**
The first step of the SafetyNet Attestation check is that the server sends a nonce. By sending this nonce, it allows the SafetyNet Attestation evaluation to be unique and therefore replay attacks can be prevented. The nonce provided by the server needs to be fresh and at least 16 bytes in length.

2. **Android application calls SafetyNet Attestation API**
The SafetyNet Attestation API on the Android device is part of the Google Play services. In order to call this API, the Android device needs to satisfy the following requirements:
   - Android device needs to be able to connect to Google’s server.
   - Android device needs Google Play services version 13.0 or above.

Because of those requirements, the developer should always check whether those requirements are met before proceeding. If an incorrect Google Play services version is installed, the Android application might stop responding after calling the SafetyNet Attestation API. Furthermore, when the requirements are satisfied, the Android application performs the API call with the provided nonce from its application server.

3. **SafetyNet Attestation service evaluates and requests an attestation**
Once the SafetyNet Attestation API is called, the Android device’s environment and Android application which is running will be evaluated outside the Android application’s sandbox. This is performed by the Google Play services process. Unfortunately, Google does not specify what and how these checks are performed. However, Kozyrakis posted a presentation in which it claims that the SafetyNet Attestation check’s binary is dynamically loaded from the Google server [84]. The result of the SafetyNet Attestation evaluation will be forwarded to Google’s server.
4. **Google’s server sends the signed attestation**
The Google server analyzes the information provided by the device’s SafetyNet Attestation service and sends the attestation back to the Android device’s SafetyNet Attestation API. This result is sent in JWS format; this format is defined in RFC7515 [85] and, comprises three components:

- JOSE header: Comprises parameters about cryptographic algorithms and further details that are used to ensure the integrity of the provided JWS. This information is needed to validate the JWS signature.
- JWS payload: The actual data.
- JWS signature: The digital signature over the data.

5. **SafetyNet Attestation API forwards signed attestation**
Once the SafetyNet attestation API received the signed attestation response from Google’s server, it will forward this result back to the Android application.

6. **Android application forwards signed attestation**
Once the Android application received the signed attestation response from the SafetyNet Attestation API, it will forward this result to the Application server.

7. **Server validates the signed attestation and makes anti-abuse decisions**
The received signed attestation needs to be validated on its integrity. This can be done by using Google’s signing certificate. Google’s certificate can be used to authenticate the signed attestation which is in JWS format. Furthermore, in the payload of the attestation is a nonce; the application server needs to verify whether this nonce matches the nonce which was initially sent in step 1. By validating the nonce, replay attacks of SafetyNet’s attestation can be prevented.

When the signature and the nonce in the payload is correct, the server can make anti-abuse decisions. This can be done by evaluating the payload of the provided attestation: section 5.2.3 describes the properties of SafetyNet’s attestation in more detail. The anti-abuse decision depends on what is considered to be acceptable by the Android developer. Finally, the server sends its decision back to the Android application. For instance, when certain properties are not met, the application server can send an error message and terminate the TLS session.

5.2.3 **SafetyNet’s attestation payload**
This section describes SafetyNet’s attestation payload. This payload, which is part of the JWS format, is generated by the SafetyNet Attestation service on the Android device (step 3) and signed by Google’s server (step 4). Based on the provided payload (step 6), the application server can make anti-abuse decisions and communicate this back to the Android application (step 7). SafetyNet Attestation’s workflow is illustrated in figure 10. An example of a SafetyNet’s attestation payload in JSON format is illustrated in figure 11. Below follows an explanation for each property of SafetyNet’s attestation payload [6].
The timestampMs is a representation of time in UNIX epoch format, which is about when the signed attestation was generated, in JWS format, by Google’s server. This timestamp can be compared to an acceptable time frame based on when the server sent the nonce to the Android application (step 1). The goal of the timestampMs field is to prevent replay attacks.

nonce
The nonce field represents a single-use token that should be equal to the initial nonce which was provided by the server (step 1). When the value in the attestation is not the same as the initial value provided by the server, then the signed attestation should not be accepted by the server. The goal of the nonce field is to make sure that the request (step 1) and response (step 7) match to prevent replay attacks.

apkPackageName
The apkPackageName represents the calling Android application’s package name. The server can use this field to compare it to an expected value. When those two values do not match, it may indicate that the server is not communicating with the genuine Android application. The apkPackageName field is focused on the integrity of the Android application.

apkDigestSha256
The apkDigestSha256 contains a base-64 encoded SHA-256 hash of the APK which is installed on the device. However, distribution channels such as the Google Play store may include additional metadata into the APK. Because of this, apkDigestSha256 might differ from the hash value of the APK which was originally distributed by the developer. Furthermore, the properties of apkPackageName and apkCertificateDigestSha256 are sufficient to assess the integrity of the Android application [86]. Since those two properties are sufficient, Google recommended in February 2018 to use those two properties instead to verify the integrity of the Android application [87]. Also, apkDigestSha256 is not described in the SafetyNet Attestation document [6]; this property was observed during the development of an Android application which is described in section 6.2.

apkCertificateDigestSha256
An Android application needs to be signed to be installed on an Android device. The apkCertificateDigestSha256 contains a base-64 encoded SHA-256 hash of the calling Android application’s signing certificate(s). The goal of this data field is to compare it to an or multiple expected value(s). If the values do not match, it may indicate that the Android Application is tampered with and, was for instance signed by the attacker’s certificate(s). The apkCertificateDigestSha256 field is focused on the integrity of the Android application.
ctsProfileMatch
The ctsProfileMatch data field is a boolean. This data field represents an integrity verdict of the Android device which is running the Android application. The value of ctsProfileMatch is true when the Android device is certified, and the profile of the device matches the profile of a device that has passed the Android compatibility testing. An Android device is certified when Google tested the device for security and performance and has preloaded Google applications [88]. Otherwise, the value of ctsProfileMatch is false. In table 3 follows some examples of scenarios and the value for ctsProfileMatch. The value for ctsProfileMatch is true unless one of the described scenarios applies in which the value is false. In addition, more scenarios may exist but are not described in SafetyNet’s documentation.

basicIntegrity
The basicIntegrity data field is a boolean. This data field represents an integrity verdict of the Android device which is running the Android application. However, basicIntegrity is more lenient, compared to ctsProfileMatch. The value of ctsProfileMatch is true when it is likely that the Android device is not tampered with. However, it did not necessarily pass the Android compatibility test. In table 3 follows some examples of scenarios and the value for basicIntegrity. The value for basicIntegrity is true unless one of the described scenarios applies in which the value is false. In addition, more scenarios may exist but are not described in SafetyNet’s documentation.

<table>
<thead>
<tr>
<th>Device status</th>
<th>Value ctsProfileMatch</th>
<th>Value basicIntegrity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Certified, genuine device that passes Android’s Compatibility Test Suite</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Certified device with unlocked bootloader</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>Genuine but uncertified device, for instance, when the manufacturer does not apply for certification</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>Device with custom ROM, but not rooted</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>Android emulator</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>No device, such as a protocol emulating script</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Signs of system integrity compromise, one of which may be rooting</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Signs of other active attacks, such as API hooking</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

Table 3: Examples SafetyNet’s values for ctsProfileMatch and basicIntegrity [6]

5.3 SafetyNet Attestation implementation
This section describes a brief explanation about how the SafetyNet Attestation service can be implemented in an Android application; the process of the SafetyNet Attestation service is explained in section 5.2.

The first step in the Android application for the SafetyNet Attestation check is to request a nonce from its application server. Using this nonce, the SafetyNet Attestation evaluation can be initiated. Figure 12 illustrates a simple example of this process. On line 2 the SafetyNet Attestation API is called to start the evaluation. This is done by calling the attest() function on the main thread using the provided nonce by the server and the API key. Since the request is sent to Google’s server, via the Google Play services, there is no intermediate response. Therefore, on line 4, it needs to wait for Google’s response which contains the signed attestation. Finally, once the response is received, the signed attestation can be retrieved using the getJwsResult() function; this is illustrated on line 6.
The request for the nonce and sending of the signed attestation to the application server is out of the scope of this section; these processes can be performed in many different ways. Hence, the signed attestation on line 6 needs to be communicated back to the application server.

```java
//Initiate the SafetyNet Attestation process on the main thread using the provided nonce and API key
Task<SafetyNetApi.AttestationResponse> safetyNetTask = SafetyNet.getClient(thread).attest(nonce, apiKey);
//Wait for SafetyNet's process to be finished
SafetyNetApi.AttestationResponse responseValue = Tasks.await(safetyNetTask);
//Get the signedAttestation, in JWS format, from SafetyNet's response
String signedAttestation = responseValue.getJwsResult();
```

Figure 12: Example SafetyNet Attestation implementation [6]
6 Test environment setup

This chapter describes the setup for the test environment which is used to assess the experiments performed in chapters 7, 8 and 9. Section 6.1 gives an introduction about the test environment and explains what it is used for. The test environment consists of an Android application and its application server, those are explained in more detail in section 6.2.

6.1 Introduction

To get a better understanding of certificate pinning and the SafetyNet Attestation service, a test environment is developed during the course of this research project. The test environment consists of an Android application and its application server. The test environment is also used to assess the experiments which are described in chapters 7, 8 and 9.

The Android application is called the “TLSConnector” and is developed in Java. The TLSConnector supports certificate pinning for two APIs: Android’s default library and the OKHttp library version 4.2.1. These two libraries are discussed in section 4.3. The goal of the TLSConnector is to initiate the SafetyNet Attestation check using a certificate pinning library for the requests. The SafetyNet Attestation service is described in more detail in chapter 5. Since the TLSConnector is using the SafetyNet Attestation service, it also requires a server implementation to validate SafetyNet’s signed attestation. The server implementation is called the “TLSConnectorServer” and is developed in PHP. The TLSConnector and the TLSConnectorServer are explained in more detail in section 6.2.

6.2 TLSConnector

The TLSConnector supports two certificate pinning libraries and the SafetyNet Attestation service. The SafetyNet Attestation service requires, as described in section 5.2.2, Google Play services version 13.0 or above. Therefore, the first thing the TLSConnector performs during startup is a check to determine whether this requirement is satisfied. When that is the case, a success message is printed; otherwise, an error message.

When the Google Play services version check passed, the user can select a certificate pinning library for the requests to perform the SafetyNet Attestation check; those requests are processed using the TLSConnectorServer. The Android’s default library and the OKHttp library are items in the dropdown list in the user interface, in which the user can choose from. Also, the user can choose whether it wants to pin the correct certificate. Figure 13 illustrates a screenshot of this process.

Once all the options are set by the user, it can choose to start the SafetyNet Attestation check, using the certificate pinning library for the requests, by pressing the button in the user interface. This will initiate the following process:
1. TLSConnector sends a GET request to the TLSConnectorServer in order to receive a nonce.
2. TLSConnectorServer generates a nonce, stores it, and returns it to the TLSConnector.
3. TLSConnector calls the SafetyNet Attestation API with the provided nonce.
4. SafetyNet Attestation API returns a signed attestation to the TLSConnector.
5. TLSConnector sends a POST request to the TLSConnectorServer to provide SafetyNet’s signed attestation.
6. TLSConnectorServer verifies and analyzes the provided signed attestation, including the provided nonce in step 2, and returns the final result of the assessment to the TLSConnector.

Note that the process above is similar to the process illustrated in figure 10. For the requests in step 1 and 5, certificate pinning is enforced using the chosen library from the dropdown list. When the user chose to pin the correct certificate, it will continue to perform the process as described above. Otherwise, after step 1, the process will be terminated, and an error message will be printed since the pinned certificate did not match the TLSConnectorServer’s certificate. Figure 14 and 15 illustrates a screenshot of the happy path for the SafetyNet Attestation check, using Android’s default library and the OKHttp library, on the TLSConnector.

<table>
<thead>
<tr>
<th>TLSConnector</th>
<th>TLSConnector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Certificate pinning library</td>
<td>Certificate pinning library</td>
</tr>
<tr>
<td>Android’s default library</td>
<td>OKHttp library</td>
</tr>
<tr>
<td><img src="checkmark" alt="Pin correct certificate" /></td>
<td><img src="checkmark" alt="Pin correct certificate" /></td>
</tr>
<tr>
<td>PERFORM SAFETYNET ATTESTATION CHECK</td>
<td>PERFORM SAFETYNET ATTESTATION CHECK</td>
</tr>
</tbody>
</table>

**Output**
- Google play services - Availability and version check
- Success: Device has Google Play service version 13+ installed
- Android’s default library - Pin certificate
- Certificate for host cees.nvlab.nl is pinned
- Android’s default library - Request nonce
- URL: https://cees.nvlab.nl/index.php/api/getnonce
- Pinned certificate: correct
- Nonce: 7AqG4AmzjZ2dL+HwVoA6jZaQlRkX6hDYaqVUKs5yFho=
- SafetyNet Attestation - SafetyNet check
- Signed Attestation received
- Android’s default library - Send signed attestation
- URL: https://cees.nvlab.nl/index.php/api/validatejws
- Pinned certificate: correct
- Result: Android SafetyNet Attestation check passed.

**Figure 14:** TLSConnector SafetyNet Attestation process using Android’s default library

**Output**
- Google play services - Availability and version check
- Success: Device has Google Play service version 13+ installed
- OKHttp library - Pin certificate
- Certificate for host cees.nvlab.nl is pinned
- OKHttp library - Request nonce
- URL: https://cees.nvlab.nl/index.php/api/getnonce
- Pinned certificate: correct
- Nonce: 7AqG4AmzjZ2dL+HwVoA6jZaQlRkX6hDYaqVUKs5yFho=
- SafetyNet Attestation - SafetyNet check
- Signed Attestation received
- OKHttp library - Send signed attestation
- URL: https://cees.nvlab.nl/index.php/api/validatejws
- Pinned certificate: correct
- Result: Android SafetyNet Attestation check passed.

**Figure 15:** TLSConnector SafetyNet Attestation process using the OKHttp library
The TLSConnectorServer verifies and analyzes the SafetyNet’s signed attestation in step 6. In this process, the server verifies the signature of the signed attestation and compares the payload to a set of expected values; figure 16 illustrates the expected values of the TLSConnectorServer.

```java
// Expected parameters value for application for SafetyNet's attestation validation
package com.example.tlsconnector;
public class TLSConnectorServer {
    public static void main(String[] args) {
        String packageNameExpected = "com.example.tlsconnector";
        String certificateFingerprintExpected = "11e4f9c5754d5d4c10144a02b976998c9b5a7c01bd0e220517b6f78a52d97a2b";
        String osVersionExpected = "true";
        String basicIntegrityExpected = "true";
    }
}
```

**Figure 16: TLSConnectorServer expected values SafetyNet’s attestation check**

The source code of the TLSConnector and the TLSConnectorServer can be found in a public repository on GitHub [89, 90]. The TLSConnector and the TLSConnectorServer are used for the experiments in chapters 7, 8 and 9.
7 Bypassing certificate pinning

This chapter discusses experiments to bypass certificate pinning in an Android environment. In the past, Northwave bypassed certificate pinning with a tool called the Android-SSL-Trustkiller. In order to get a better understanding of the Android-SSL-Trustkiller, this tool is analyzed and discussed in section 7.1. Based on the analysis in section 7.1, an alternative framework was found. This framework is called Frida and is discussed in more detail in section 7.2. Finally, section 7.3 is dedicated to experiments to bypass certificate pinning in an Android environment using the Frida framework.

7.1 Introduction

The Android-SSL-Trustkiller is a tool that was Northwave’s solution to bypass certificate pinning [5]. Using this tool, it was possible to deploy a man-in-the-middle position. The Android-SSL-Trustkiller is using the Cydia Substrate platform [91], which can be used for code modification at run-time. The Cydia Substrate framework works for iOS on versions 2.0 through 9.1 and Android versions 2.3 through 4.3. Furthermore, in order to use Cydia Substrate, the device needs to be jailbroken (iOS) or rooted (Android). Thus, to make Android-SSL-Trustkiller working, the Cydia Substrate platform needs to be deployed on a rooted Android device. Android devices are Linux-based and do by default not have the superuser binary in the filesystem. Rooting an Android device simply adds this binary so that the user can perform actions with root (superuser) privileges. Section 8.2 describes the rooting process of Android devices in more detail.

The Android-SSL-Trustkiller hooks and overrides function at run-time of the following classes: HttpsURLConnection, TrustManagerFactory, SSLContext and SSLSocketFactory. Furthermore, it also injects a custom implementation of the TrustManager interface. By doing this, the certificate pinning of the targeted Android application is disabled. This results in that the Android application sets up a TLS connection without certificate pinning. Because of this, the certificate validation of the TLS handshake falls back on the device’s Keystore.

Since the Android-SSL-Trustkiller falls back on the device’s Keystore, the CA certificate, used to deploy a man-in-the-middle attack, needs to be added to this Keystore. The desktop application which is used by Northwave’s Red team to achieve a man-in-the-middle position is called Burp Suite [92]. Burp Suite allows a penetration tester to set up a web proxy that can be used to intercept, inspect and modify raw traffic passing in both directions. Burp uses its own CA certificate which signs other certificates, generated on the flow, for its proxy connections. By adding Burp’s CA certificate to the device’s Keystore, Burp’s man-in-the-middle position will be considered as trusted by the device; all the host’s certificates are signed by Burp’s CA certificate [93]. Appendix 15.1 describes how Burp Suite can be deployed on the desktop and the Android device.

To conclude, the Cydia Substrate framework is not supported for Android versions above 4.3. Also, as described in section 4.2, for Android versions above 6.0, user-added CA certificates are not considered as trusted anymore by default. Therefore, Burp’s CA certificate cannot be added to the Keystore to be considered as trusted. Those two reasons are why the Android-SSL-Trustkiller, in combination with Burp suite, does not work anymore. Moreover, the Android device needs to be rooted, this is another problem when the Android application is using the SafetyNet Attestation service. Bypassing the SafetyNet Attestation service is discussed in section 8.3.
7.1.1 Alternative

The Android-SSL-Trustkiller is using the Cydia Substrate framework to hook and change functions at run-time, this section describes an alternative framework to do this.

During this research project, the Frida framework was found as an alternative. Frida is a framework that can, like the Cydia framework, hook and modify functions at run-time in an Android environment. However, compared to the Cydia framework, Frida supports both rooted and nonrooted Android devices [94]. Section 7.2 gives a detailed description of the Frida framework. Furthermore, section 7.3 describes the experiments that were performed using this framework to bypass certificate pinning.

7.2 Frida

Frida framework is, like the Cydia framework which is being used for the Android-SSL-Trustkiller, a framework that can be used to deploy code modification at run-time into an application [95]. The code modification with Frida can be deployed by injecting snippets of JavaScript into native applications on Windows, macOS, Linux, iOS, Android and QNX environments [96]. Frida is compatible with Android devices with the ARM, ARM64, X86 or X86-64 architecture. At the time of writing (May 2020), Frida is being updated regularly [97]. Compared to the Cydia framework, Frida supports both rooted and nonrooted Android devices [94]. Section 8.2 explains the process of rooting Android devices in more detail.

Frida is written in the programming language C and injects Google’s V8 engine into the targeted process. The V8 engine can be used to execute JavaScript source code and is developed to run standalone or to be embedded into any C++ application [98]. When the V8 engine compiles the JavaScript source code, the parser generates an abstract syntax tree which is a representation of the syntactic structure of the code. The interpreter of the V8 engine generates the bytecode from this syntax tree. With the generated byte code, the V8 engine generates the machine code which can be directly executed by the machine that it is running on [99].

Frida injects the V8 engine to deploy code modification with full access to the device’s memory; this privilege enables to hook functions and calling native functions inside the targeted process. Frida leverages the targeted process by deploying a bi-directional communication channel between the application’s process and the engine which is running the code modification script [96].

Figure 17 illustrates a simple example of a code modification script for an Android application. This script overrides the function `openConnection()`, which is defined in the URL class; figure 7 illustrates an example of how this function is used in practice. The code modification script hooks the function on line 7, prints text in the console on line 8, and returns its regular behavior as a result on line 9. Appendix 15.2 describes a step-by-step guide about how this script can be injected into a process on an Android device [100].

![Figure 17: Example Frida code modification script](image-url)
In order to make Frida working on an Android device, it needs to be deployed on the desktop and device. Frida works with the client-server principle in which the desktop (client) sends the code modification script to the device (server). On the Android device, the Frida server must be running to inject the script into a process and leverages functions with the bi-directional communication channel. On a rooted Android device, Frida’s server can be executed with root privileges; on a nonrooted device, Frida needs to be injected into the source code of the targeted Android application. Both scenarios are discussed below in more detail.

7.2.1 Rooted device
This section describes the deployment of the Frida framework on a rooted Android device. To deploy Frida, the adb command-line tool can be used to access the device’s UNIX shell. Using adb, the Frida server needs to be pushed into the device’s file system. Once it is located in the device’s file system, it needs execute privileges; this can be done with the (root) superuser. Finally, the Frida server needs to be executed as a background process with root privileges. This can be achieved by executing the Frida server as superuser in the device’s UNIX shell. The step-by-step guide of this process is described in appendix 15.3 [94].

Once the Frida server is running as a background process with root privileges, it can target any active process on the Android device. To get an overview of the processes which are currently running, the following Frida command can be executed on the client:
frida-ps -U

This command sends the request, over the USB cable, to the Frida server on the device. Finally, the server sends the result back to the client which prints it in client’s console. Figure 18 illustrates an example of an output of the described command. In this list, the package names of actively running Android applications can be found; for instance, TLSConnector’s package name is com.example.tlsconnector.

7.2.2 Nonrooted device
This section describes the deployment of the Frida framework on a nonrooted Android device. Since the Android device does not have root access, Frida cannot be executed as a privileged background process as discussed in section 7.2.1. Instead, Frida needs to be injected into the Android application. This can be achieved by modifying the installation file of the targeted Android application. Therefore, an important requirement for this process is, to be in possession of the application’s installation file; as discussed in section 4.1 this file has the suffix .apk. The step-by-step guide of this process is described in appendix 15.4 [101]; below follows a summary.
The first step to inject Frida into an Android application, is to decompile the APK using the apktool; the apktool is part of the Android SDK [102]. This will create a folder of the decompiled Android application. In this folder, the Frida Gadget needs to be copied into; Frida Gadget is a Frida server that is made to be injected into an Android application as a third-party library [103]. However, to make this effective, a reference to Frida's library needs to be made. This can be achieved by finding the starting point of the application, by analyzing the decompiled source code, and to add a reference to the Frida library. Once this reference is made, the application needs to be recompiled using the apktool, this will create a new APK file. Furthermore, this file needs to be aligned and signed using the apksigner; the apksigner is part of the Android SDK [104]. Finally, the signed APK needs to be installed using the adb command-tool.

When the modified Android application is executed, the screen will stay white, until the Frida client on the desktop sends a code modification script to the Frida Gadget which is running inside the application. Hence, the white screen means that it is waiting for a connection with a Frida client. Unlike with the deployment of the Frida server on a rooted device, the Frida client can only inject a code modification script into the modified Android application. As described in section 7.2.1, the following command can be sent to the Frida server to obtain an overview of all active processes:

```
frida-ps -U
```

The outcome of this command is illustrated in figure 19; only the Frida Gadget can be targeted to inject a code modification script from the client. The Gadget process is the modified Android application. This result is different compared to the result for the deployment of Frida on a rooted Android device which is illustrated in figure 18.

### 7.3 Bypassing certificate pinning

This section covers the results of several experiments with as goal to bypass certificate pinning in an Android environment. These experiments are performed using the Frida framework. The experiments are tested on:

- Northwave’s Nexus 5X running LineageOS version 15.0 (based on Android 8.0) with root privileges
- OnePlus 3T running its stock Android version 9.0 with root privileges
- Xiaomi Mi 9T running its stock Android version 9.0 without root privileges

Using Frida, a code modification script can be injected at run-time into an Android application; the details of this framework are described in section 7.2. Using the information and analyses in previous chapters, code modification scripts, compatible with Frida’s framework, are developed. This section covers those code modification scripts as part of the experiments. The goal of the scripts is to bypass certificate pinning, in combination with Burp suite, so that a man-in-the-middle position can be achieved. Section 7.1 and appendix 15.1 describe Burp suite and the deployment of it in more detail.

The code modification scripts are focused on bypassing Android’s default library and the OKHttp library. Both of those libraries are implemented in the TLSConnector to support the certificatepinning mechanism. Therefore, all the experiments in this section are assessed with
Bypassing certificate pinning

the TLSConnector which is described in chapter 6. Since the code modification scripts are tested on the TLSConnector, it is targeting OKHttp library version 4.2.1. The development of the code modification script is split into three phases:

- Bypass Android's default library
- Bypass OkHttp library
- Information logging of SafetyNet Attestation service

Information logging of the SafetyNet Attestation service gives convenient insights about when the SafetyNet Attestation API is called and what is returned in the payload of the signed attestation. This section is split into the three phases described above. Finally, section 7.3.4 describes the final result of those three phases in which the developed code modification scripts are merged into one script.

7.3.1 Bypass Android’s default library

Android’s default library is using classes and functions which are part of the Android operating system. An important interface for this approach is the TrustManager interface. This interface needs to be implemented to manage trust decisions, such as certificate chain validation, for the setup of a TLS session. The TrustManager is implemented and hooked in the Android-SSL-Trustkiller’s solution [5]. Recall figure 8 in section 4.3, which illustrates the implementation of certificate pinning using Android’s default library, wherein an SSLContext instance is initialized using Android’s default Trustmanager implementation.

The first step in this experiment, was to look into existing code modification scripts for Frida. An interesting article, written by Vinci [105], was found which describes a script to bypass certificate pinning by hooking and overriding Android’s default TrustManager implementation. However, after testing this script, it was concluded that it was not working. Nonetheless, it showed the name of Android’s default implementation for the TrustManager interface, which is the TrustManagerImpl class. After investigating, the source code, including Javadoc, of TrustManagerImpl class was found; the source code is published by Google on GitHub [106]. Vinci’s code modification script is hooking the function checkTrustedRecursive(). According to Google’s Javadoc, this function is responsible for recursively building the certificate chain until a valid chain is found. Furthermore, this function returns the entire valid certificate chain. Vinci’s script hooks this function and simply returns an empty list of certificates; hence, an empty certificate chain.

The first parameter of the function checkTrustedRecursive() is certs, this is an array of X.509 certificates which are provided by the peer. This means that any certificate might be accepted by directly returning the value of this parameter as the return value for this function. Hence, by directly returning the value of certs, the provided certificate chain by the peer will be considered as trusted. Based on this finding, a code modification script was developed; the result is illustrated in figure 20. On line 6 the function checkTrustedRecursive() is hooked, the first parameter certs is an array of X.509 certificates. However, the function returns a different list type, called an ArrayList. Therefore, cert is copied into an ArrayList; this ArrayList is initialized on line 10. The for-loop, on line 12, iterates over the X.509 certificates in the array and copies it into the ArrayList on line 13. Finally, this ArrayList is returned as return value for the function checkTrustedRecursive() on line 15.

The developed code modification script was tested on the TLSConnector. This test was successful. By deploying this script, together with Burp Suite, a man-in-the-middle position
Bypassing certificate pinning can be achieved. This script allows that any certificate is accepted for a TLS session. Hence, also Burp’s certificates.

Recall Android’s default library implementation for certificate pinning in figure 8. On line 24 the function init() is called, this function is defined in the SSLContext class. The second parameter of this function is an array of TrustManager interfaces; these instances are explained in more detail in section 4.3. TrustManager is an interface and the TrustManagerImpl class is Android’s default implementation for it [106]. However, it is up to the developer to use Android’s default TrustManagerImpl or to use another implementation. In case the developer chooses not to use Android’s default implementation, the code modification script, illustrated in figure 20, will not work since it is specifically hooking the TrustManagerImpl class.

Therefore, an additional code modification script was developed which is illustrated in figure 21. On lines 7-9 a local KeyStore is initialized as illustrated in figure 8. This instance is used to initialize an instance of TrustManagerImpl which is stored in an array on line 10. Furthermore, on line 13 the function init() is hooked and its regular behavior is returned as result of the function on line 16; however, the second parameter of this function call is replaced with the TrustManagerImpl instance which was initialized on line 10. By doing this, it is ensured the TLS session will be build using the TrustManagerImpl class. The combination of the code modification scripts, illustrated in figures 20 and 21, Android’s default library can be bypassed; this is tested on the TLSConnector.

---

**Figure 20:** Bypass Android’s default library by hooking checkTrustedRecursive()

**Figure 21:** Bypass Android’s default library by hooking init()
7.3.2 OKHttp library
The code modification scripts which were developed for Android’s default library, illustrated in figure 20 and 21, are also tested on the OKHttp library by using the TLSConnector. Unfortunately, this test was not successful. However, an interesting finding of this test was that the function checkTrustedRecursive(), which is hooked with the code modification script in figure 20, is called by the OKHttp library during the TLS handshake. Hence, it seems that the OKHttp library is using Android’s default implementation of the TrustManager interface, which is the TrustManagerImpl class, to make trust decisions. This finding indicated that this code modification script might be necessary as part of the solution to bypass the OKHttp library. However, this was not the complete solution according to the test which was performed with the TLSConnector.

Recall the OKHttp library implementation for certificate pinning in figure 9. On line 5 the host and the hash of the certificate are set using the add() function; this function is part of the Builder class which is nested in the CertificatePinner class. The source code of the OKHttp library is posted on GitHub[107]. OKHttp’s source code supported the development of a code modification script to bypass the OKHttp library. Figure 22 illustrates the result of this script. On line 6, the function add() of CertificatePinner’s Builder class is hooked, and on line 10 the regular behavior is returned. However, the first parameter, which is the host parameter, is replaced by a fake hostname which is initialized on line 9. This results in that the certificate is not pinned for the host which it was initially meant for.

```
function() {
    // Class config

    // Override hostname in certificate pin builder to a fake hostname
    CertificatePinnerBuilder.add.implementation = function(host, pin) {
        console.log("OKHttp: remove pin for host: "+ host);
        var fakeHostName = "x";
        return this.add(fakeHostName, pin);
    }
}
```

Figure 22: Bypass OKHttp library by hooking add()

The code modification script which is illustrated in figure 22 is tested on the TLSConnector. However, it does not give the full solution to bypass certificate pinning for the OKHttp library. The OKHttp library is using the TrustManagerImpl class which is using the function checkTrustedRecursive() to recursively construct the certificate chain until a valid chain is found. This will not succeed when Burp Suite is used, Burp’s CA certificate is not considered as trusted by the Android operating system. Therefore, the combination of the code modification scripts, illustrated in figure 20 and 22, were tested on the TLSConnector. The combination of those two scripts resulted in successfully bypassing the OKHttp library; all certificates provided by the peer are considered as trusted.

7.3.3 Information logging Android SafetyNet Attestation service
SafetyNet’s signed attestation is provided by Google in JWS format [85]. Hence, the attestation which is returned by Google’s server is only signed, not encrypted. This allows hooking the SafetyNet’s API functions which are responsible for starting the SafetyNet Attestation evaluation, and to get Google’s response containing the payload of the attestation in plain text. Moreover, since it is signed by Google, it is not possible to alter the signed attestation; the signature will be incorrect when the information in the payload is altered. Note, when an application server is not validating the signature of the signed attestation, this can be a big risk since in that case the signed attestation can be altered.
Figure 23 illustrates the code modification script to hook the SafetyNet Attestation API; the functions which are hooked in this script are explained in section 5.3 and illustrated in figure 12. On line 9, the function `attest()` is hooked which is responsible for initiating the SafetyNet Attestation evaluation. This function is a member of the `SafetyNetClient` class. On line 11, a message is printed in the client’s console that SafetyNet’s evaluation function is called to initiate the check process. On line 12 its regular behavior is returned as part of the function.

After Google Play services and Google’s server processed the evaluation, a signed attestation is returned by google. The Android application can get this result by calling the `getJwsResult()` function which is a member of the `AttestationResponse` class. This class is nested into the `SafetyNetApi` class. The function `getJwsResult()` is hooked on line 16. On line 18, its regular behavior is called and stored into a variable. The signed attestation, which is in JWS format, is dot-separated with having the payload in the second part [85]. This payload is retrieved from the signed attestation, which was stored into a variable, and the result is stored into a variable on line 19. The payload is base64 encoded, therefore on line 20 this payload is decoded and copied into a string which is then stored into a variable; this variable holds the payload of the signed attestation in plain text. On line 21, this payload, in plain text, is printed into the client’s console. Finally, on line 22 its regular behavior, which was initially called and stored into a variable on line 18, is returned for the hooked function.

```java
Java.perform(function() {
  // Classes config
  var SafetyNetClient = Java.use("com.google.android.gms.security.trust.SafetyNetClient");
  var AttestationResp = Java.use("com.google.android.gms.security.trust.SafetyNetApi#AttestationResponse");
  var Base64 = Java.use("java.util.Base64");
  var String = Java.use("java.lang.String");

  // Logs when SafetyNet Attestation evaluation starts and continues its regular behavior
  SafetyNetClient.attest.implementation = function(nonce, apiKey)
  {
    console.log("SafetyNet Attestation is called");
    return this.attest(nonce, apiKey);
  }

  // Logs the payload of SafetyNet Attestation evaluation and continues its regular behavior
  AttestationResp.getJwsResult.implementation = function()
  {
    var signedAttestation = this.getJwsResult();
    var payloadEncoded = (signedAttestation.split(\"\"))[1];
    var payloadDecoded = String.fromCharCode(Base64.getDecoder().decode(payloadEncoded));
    console.log("SafetyNet Attestation payload\n" + payloadDecoded);
    return signedAttestation;
  }
});
```

Figure 23: Information logging SafetyNet Attestation API by hooking `attest()` and `getJwsResult()`

The code modification script was tested on the TLSConnector. Using this script, information about when the SafetyNet Attestation API is called will be printed into the client’s console, and its response, which is the signed attestation, will also be printed. Note that this script is not changing the behavior of the SafetyNet Attestation service but is purely for logging purposes. This script allows to get better insights about the SafetyNet Attestation service in Android applications other than the TLSConnector.
7.3.4 Android-CertificatePinning-Killer

In sections 7.3.1, 7.3.2 and 7.3.3 multiple code modification scripts are described which can be used to bypass Android’s default library and OKHttp library and, to perform information logging of the SafetyNet Attestation service. To make those script effective, the scripts are merged into one code modification script, this solution is called the Android-CertificatePinning-Killer. The Android-CertificatePinning-Killer is uploaded into a public repository on GitHub [108].

In order to make the Android-CertificatePinning-Killer scalable, each code section that focuses on a library is surrounded by a try-catch block. This prevents termination of the script when a library or function could not be found. For instance, when the Android-CertificatePinning-Killer is injected into an Android application that does not use the OKHttp library, the script will fail without a try-catch block; in that case, Frida was not able to hook the library. With a try-catch block around each library’s code section, only the code sections in the try-catch block will potentially fail, instead of the script as a whole. Furthermore, try-catch blocks give the possibility to print an error message in the console when the library was not found. Figure 24 illustrates how the try-catch blocks are formatted in the Android-CertificatePinning-Killer. When the library x is found, a success message will eventually be printed in the console. Otherwise, when the library is not found, a fail message will be printed; in both scenarios, the Android-CertificatePinning-Killer will continue executing.

```java
function () {
    try {
        console.log("Looking for library x");
        //Hook library X and its functions
        console.log("Library x was successfully hooked");
    } catch (e) {
        console.log("Library x was not found");
    }
}
```

*Figure 24: Format try-catch block for code modification script*

The Android-CertificatePinning-Killer is tested, with the try-catch blocks included, on the TLSConnector. Figure 25 illustrates the messages which are being printed in the client’s console by the script. In the first part of the script deployment, messages are printed in the console which indicates that the libraries are successfully hooked. After this, the SafetyNet Attestation check is executed on the TLSConnector, this results in that the certificate pinning, in this case Android’s default library, is bypassed. Furthermore, as part of this test, Burp Suite is used to deploy a man-in-the-middle position. With Burp Suite, the communication between the TLSConnector and its server can be observed in plain text. Figure 26 illustrates the nonce, sent from the TLSConnectorServer, in plain text.

The experiment of the Android-CertificatePinning-Killer also resulted in an additional finding. In the logging of the SafetyNet Attention’s payload, the field `advice` was observed; this is illustrated in figure 25. This field is not mentioned in the documentation of the SafetyNet Attestation service. It seems that this field contains information about why the SafetyNet Attestation check failed. This test was executed on Northwave’s Nexus 5X which does not pass the SafetyNet Attestation check; SafetyNet’s property `ctsProfileMatch` is false. The reason why the Nexus 5X failed the SafetyNet Attestation check, is discussed in more detail in section 8.3.1. Furthermore, based on this finding, the field `advice` is added to the TLSConnectorServer to be printed as part of the SafetyNet Attestation check’s result.
Bypassing certificate pinning

Figure 25: Android-CertificatePinning-Killer output in the console

Figure 26: Android-CertificatePinning-Killer nonce response in Burp Suite
8 Bypassing SafetyNet Attestation service
This chapter discusses experiments to bypass the SafetyNet Attestation service. Section 8.1 covers an introduction about this topic wherein it explains why the SafetyNet Attestation check might fail on an Android device. One of the reasons why the SafetyNet Attestation check can fail is when an Android device is rooted. To get a better understanding of what a rooted Android device is, this topic is explained in detail in section 8.2. Finally, section 8.3 is dedicated to the experiments to bypass the SafetyNet Attestation service.

8.1 Introduction
The SafetyNet Attestation service focuses on the integrity of the Android application and the device on which the application is running on. The SafetyNet Attestation service is explained in more detail in chapter 5. To prevent replay attacks, SafetyNet's attestation comprises the properties nonce and timestampMS. To ensure the integrity of an Android application, SafetyNet's attestation comprises the properties apkPackageName and apkCertificateDigestSha256 related to the application which is running on the Android device. To ensure the integrity of the Android device, SafetyNet's attestation comprises the properties ctsProfileMatch and basicIntegrity to indicate whether certain integrity conditions are met. These conditions are illustrated in table 3.

The application server can decide which properties are verified and what the expected values are. The experiments which are performed in this chapter, described in section 8.3, are using the TLSConnector to assess the result of the tests. Furthermore, the TLSConnectorServer is used to verify SafetyNet's signed attestation. The TLSConnectorServer validates the payload of SafetyNet's attestation in which it expects a certain value for the properties nonce, apkPackageName, apkCertificateDigestSha256, ctsProfileMatch and basicIntegrity. The TLSConnectorServer does not tolerate rooted Android devices and therefore the expected value for ctsProfileMatch is true; this implies that the more lenient device integrity verdict basicIntegrity is expected to be true as well. Section 8.2 describes the rooting process of Android devices in more detail.

8.2 Rooted Android Device
In section 7.1, rooted Android devices are briefly discussed. A rooted Android device is required to deploy the Android-SSL-Trustkiller and to deploy the Frida framework. Furthermore, Northwave's Red team is using a rooted Nexus 5X Android device to perform penetration tests on Android applications. During this research project, the TLSConnector is tested on the Nexus 5X. This device fails the SafetyNet's device integrity check. The analysis of why the Nexus 5X fails the SafetyNet Attestation check is described in section 8.3.1. To get a better understanding of the rooting process of Android devices, this section is dedicated to that topic.

8.2.1 Introduction
The Android operating system is, as described in section 4.1, Linux-based. Rooting an Android device is the process of allowing the user of the Android operating system to gain privileged control, which is known as root access. As the Android operating system uses the Linux-kernel as a base, rooting gives similar access as an admin; this is also known as superuser permissions. To gain superuser permissions, a rooted Android device has an executable root binary located in its filesystem. Rooting is simply adding the standard Linux function, called su, which is by default not included in the Android environment. The su binary allows to run applications with root access or to use the device's UNIX shell with root access by simply typing "su" into the command line [109].
The reason why people want to root an Android device is to get full control over their device. With superuser privileges, the user can access, alter and execute everything in the operating system. This also includes data that is being stored locally by an application. The goal of rooting is overcoming the restrictions of the manufacturer of the device. Furthermore, it can also facilitate the complete removal of the device’s operating system [7, 109].

The process to root an Android device varies per manufacturer, model and carrier. For instance, Samsung makes deals with carriers. Those carriers can decide whether it wants to release a model in which it prevents the user from rooting it. However, there are still several programs available to achieve rooting. On the contrary, the international version of LG has an unlockable bootloader which can be, when unlocked, used to install a custom Android ROM [109]. An Android ROM refers to a device’s firmware which is based on Google’s Android platform [110]. Furthermore, OnePlus is one of the more developer-friendly manufacturers, all of the company’s phones, except the T-Mobile branded OnePlus 6T, can be rooted easily without any restrictions with the unlockable bootloader [109]. Section 8.2.2 describes the bootloader in more detail. Moreover, when rooting is not allowed by the manufacturer, it is in most cases still possible to do it by using a root exploit [111].

There are over 12,000 Android device types on the market. Because of this, there is no streamlined process to root an Android device [109]. Since there is no streamlined process, the root process is not discussed in this thesis. However, as part of some rooting experiments, the rooting process for a OnePlus 3T Android device is described in section 8.3.3. Furthermore, there are two different types of root implementations, those are discussed in more detail in section 8.2.4.

8.2.2 Bootloader and flashing partitions
In section 8.2.1, the so-called bootloader is mentioned. A bootloader is a vendor-proprietary image that is responsible for bringing up the kernel on a device. The bootloader comprises many things, including the splash screen, and boots the device’s operating system [112]. Because the bootloader is an essential component of the boot process, it is stored in non-volatile memory. Furthermore, the actual implementation of the bootloader differs for each brand and model [113].

Initially, the bootloader’s state of a device is locked; this state can be changed to unlocked [114]. When the device’s bootloader is unlocked, it is possible to reflash partitions on the Android device; this includes the recovery, boot and system partition. The recovery partition stores the recovery image of the operating system and supports updates for it [115]. Since these partitions can be reflashed, it is possible to install a custom operating system on the device.

Bootloader management and flashing partitions can be managed with the fastboot mechanism. Fastboot is, like adb, a client-server tool that can be used to send commands to the Android device. The fastboot tool needs to be installed on the desktop and the Android device needs to be connected with a USB cable to this desktop. Furthermore, the Android device needs to be in the fastboot state so that it is listening to fastboot commands, which can be sent from the desktop [116]. The fastboot state can be accessed by holding a certain button combination during the startup of the device; the actual combination differs per device [117]. Using the fastboot command-tool, the user can unlock or relock the bootloader and flash partitions on the Android device. When the bootloader’s lock state changes, a factory reset will be performed to prevent unauthorized data access; this process deletes all the installed applications and user data from the device [118].
The Android operating system has a flag called \texttt{get_unlock_ability}, when this flag is set to "1", the bootloader can be unlocked in fastboot mode using a command sent from the desktop. All Google-branded devices are shipped with an Android release which has the \texttt{get_unlock_ability} flag set to "1". Moreover, when the \texttt{get_unlock_ability} flag is set to "0", the fastboot command to unlock the bootloader cannot be used immediately. In that case, the user first has to unlock the bootloader unlock command via Android's user interface; this option can be found in the settings menu under the name OEM unlocking. Once the OEM unlocking option is enabled, the bootloader can be unlocked in fastboot mode [118]. Since the Android release and its user interface can differ for each device, the manufacturer can choose not to include this functionality in the user interface; this results in that the device does not have an unlockable bootloader [109]. For instance, two popular carriers in the United States, called Verizon and AT&T, only sell carrier-branded Android devices that do not have an unlockable bootloader [119, 120, 121]. Hence, the unlockable bootloader not only differs for each Android device model but also whether it is a custom release from the carrier.

8.2.3 Verified Boot
An integral part of the boot process in an Android environment is Verified Boot. Verified Boot is a mechanism, introduced in Android version 4.4, which strives to ensure that the operating system which is running on the device is from a trusted source; this also includes security checks of all executable code and data that is part of the Android version which is being booted [122]. The Verified Boot process is part of the booting process and is enforced when the bootloader is locked. When the device’s bootloader is unlocked, it will only check whether a valid operating system was found in which the user has to accept the risk [114].

When Verified Boot is enforced, the mechanism establishes a full chain of trust for each partition and device statuses; this also includes the state of the device’s bootloader [123, 124]. This is done by calculating the hash on the fly and compare it with an expected value; this expected value is signed by the root of trust [125]. The root of trust is hardware-backed and is used to verify the expected hash values to perform integrity checks [122]. When Verified Boot detects a corrupted partition, the device will not boot or only with the user’s consent [126]. Verified Boot also supports a user-settable root of trust. When the user-settable root of trust is set, Verified Boot uses this root to cryptographically verify the partitions on the device. The user-settable root setting allows the user to install custom versions of the Android operating system without sacrificing the security controls of Verified Boot. However, it depends on the device and carrier whether the user-settable root is supported [124]. Based on the state of the bootloader and Verified Boot, the device has a verified state. The following values for \texttt{verifiedstate} are possible [114]:

- Green: Bootloader is locked, and hardware-backed root of trust is used.
- Yellow: Bootloader is locked, and user-settable root of trust is used.
- Orange: Bootloader is unlocked.
- Red (eio): Verified Boot is corrupted.
- Red (no OS found): No valid OS found.

Figure 27: Warning screen verified state yellow [114]
When the `verifiedstate` has the value green, the device boots regularly; this means that it passed all integrity checks and therefore it is not rooted. The values yellow, orange or red (eio) results in a warning screen during the startup of the device. In this warning screen, the user is notified about the `verifiedstate` and has to confirm whether it wants to continue booting. Note that when the value is orange, Verified Boot is not enforced, and it will only check whether a valid OS was found. Furthermore, when the `verifiedstate` has the value red (no OS found), the device is corrupted, and therefore the device will not boot; this results in an error screen [114]. Figure 27 illustrates an example of the warning screen when the `verifiedstate` is yellow.

8.2.4 Root implementations

In order to have an Android device rooted, an accessible root binary, also called the su binary, needs to be located in the filesystem. Where this binary is located, depends on which root implementation is used to root the device. There are two different root implementations, system root and systemless root; each implementation is discussed in this section. Section 8.3.3 goes into more detail about the root implementations by performing several experiments to bypass the SafetyNet Attestation service.

8.2.4.1 System root

System root is a root implementation in which the root binary is located in the system partition of the Android operating system. This partition mainly contains the Android framework and user space and is located under the file path `/system` on the device [127, 115]. System root also includes some init scripts in the system partition. Those scripts are executed when the Android device boots and performs a set of changes that are required to execute the root binary [128]. With system root, the root binary is typically located in either of the following locations in device’s file directory [129]:

- `/system/bin/
- `/system/xbin/`

The `bin` directory stands for binary. In a Unix-like operating system, like Android, this folder contains base executables which are required for minimal system functionality. These commands can be accessed and executed by every user on the system. This folder contains commands such as: `cat`, `cp`, `chmod`, `dir`, `dd`, `mv` and zipping tools such as `bzip` and `gzip` [130]. Furthermore, the `xbin` directory is specifically for Android and contains extra native binaries. These binaries are not essential for running the Android operating system. However, like the `bin` directory, `xbin` is accessible for all users [131].

System root can be achieved by installing a modified Android ROM on the device. This modified Android ROM contains the root binary as part of the operating system. Since every Android device has its own hardware, the compatibility of a modified ROM varies per device [127]. An alternative way to install the root binary on the device, is by installing this binary via the device’s fastboot mode. This needs to be initiated during the boot process. To reflash a partition on the device, the bootloader needs to be unlocked; this will, as described in section 8.2.2, initiate a factory reset of the Android operating system. An example of a system root library is the SuperSu library [132].
8.2.4.2 Systemless root

The systemless root is a root implementation in which the root binary is located outside the system partition of the Android operating system. With a systemless root, the boot partition is flashed with a custom image [127]. This custom boot image allows the device to boot with a fully privileged daemon from the very start of the booting process. When an application needs root access, it executes the root binary via this daemon [133]. Unlike system root, systemless root does not add or modify files in the Android's system partition or user space. Hence, the Android ROM does not have to be modified for this root implementation [127]. The root binary with systemless root is typically located in the following location on the device's file directory [129]:

- /sbin

The /sbin directory is in a Unix-like operating system, like Android, a binary folder that contains administrative tools for the operating system. Mostly, the binaries in this folder are only accessible for the root user [134]. This folder usually contains binaries essential for boot, restoring, recovering and/or repairing the system [135]. Since the systemless root daemon is running with privileged control, it can execute the commands in this binary folder. When the device is rooted, the su command is one of the executable commands in this folder.

In order to reflash a partition on the device, the bootloader needs to be unlocked; this will, as described in section 8.2.2, initiate a factory reset of the Android operating system. Hence, to reflash the boot image, the bootloader needs to be unlocked. Furthermore, Magisk is an example of a systemless root library that can be used to root an Android device for Android versions 4.2 and above. At the time of writing (May 2020), Magisk is regularly updated and is also a suite of open-source tools that can be used to gain more control over the device [136].

8.3 Bypassing SafetyNet Attestation service

This section describes several experiments to bypass the SafetyNet Attestation service. Northwave’s Android device, which is used for penetration testing, does not pass the SafetyNet Attestation check. To get a better understanding of this, section 8.3.1 describes an analysis of which operating system is running on this device, and how it is rooted. Frida Suite is also compatible with nonrooted Android devices; section 8.3.2 covers an experiment wherein Frida is embedded into an APK to make this happen. In section 8.3.3 follows several experiments that involve the two discussed rooting implementations, system root and systemless root. All the experiments in this section are assessed with the TLSConnector which is described in chapter 6 and with SafetyNet's settings as described in section 8.1.

The nonrooted and rooted experiments are performed on a OnePlus 3T Android device. For the experiments, various operating systems and root implementations are installed on this device. Based on the experiments, it can be concluded whether the SafetyNet Attestation service can detect the wrongfulness of the Android application or device integrity. Furthermore, Northwave’s Android device, the Nexus 5X, is only used to analyze why it is not passing the SafetyNet Attestation check; this is described in section 8.3.1. It was from Northwave’s side not allowed to tamper with this device.

The OnePlus 3T is by default running on OxygenOS which is OnePlus’s Android operating system release [137]. OxygenOS is close-to-stock, which means it is close to Google’s Android stock release. The difference between Google's Android release and OxygenOS are some slight user interface changes [138]. The OnePlus 3T which is used for the experiments is running on Android version 9.0. Furthermore, OnePlus distributes its official Android ROM releases on its website [139]; this enables to easily flash the default Android ROM on OnePlus 3T as part of the experiments.
8.3.1 Northwave’s Android device

Northwave’s Nexus 5X Android device is being used to perform penetration tests on Android applications for its customers. This device does, as illustrated in figure 28, not pass the SafetyNet Attestation check; it fails on SafetyNet’s property `ctsProfileMatch`. The SafetyNet Attestation service has two properties to check on device integrity, `ctsProfileMatch` and `basicIntegrity`; the Nexus 5X only triggers `ctsProfileMatch`. This means the device does not meet one or more, as illustrated in table 3, of the following properties:

- Bootloader is unlocked
- Device is not certified
- Device runs on a custom ROM

In section 7.3.4 an additional finding was observed during the test of the Android-CertificatePinning-Killer. The test revealed the property advice in the payload of the SafetyNet’s attestation. This field is added to the TLSConnectorServer to be shown as part of the result of the SafetyNet Attestation check. As illustrated in figure 28, SafetyNet’s advice is to lock the device’s bootloader. The bootloader of the Nexus 5X is indeed locked; figure 29 illustrates that the OEM unlocking function itself is disabled with the message “Bootloader is already unlocked”. Locking the bootloader of a device will, as described in section 8.2.2, initiate a factory reset. However, it is not allowed to tamper with Northwave’s Nexus 5X. Therefore, this problem could not be resolved.

Northwave’s Nexus 5X also runs on a custom ROM called LineageOS. This custom ROM, LineageOS version 15.0, is based on Android version 8.0. LineageOS is an open-source operating system that is based on Google’s original Android release; LineageOS focusses on individuality, security, longevity which includes binaries to easily root an Android device [140]. Since LineageOS is a custom ROM, this might also trigger the device integrity check of the SafetyNet Attestation service.

However, it seems that the SafetyNet Attestation check was not able to detect that the Nexus 5X is rooted; SafetyNet’s property `basicIntegrity` was not triggered. Northwave’s penetration testers were questioned about which root implementation is used to root the device, however, nobody seemed to know that. Because of that, the Nexus 5X was connected with a USB cable to a desktop, and the following adb command was executed in the device’s root directory:

```
find . -name "su" 2> /dev/null
```

In section 7.3.4 an additional finding was observed during the test of the Android-CertificatePinning-Killer. The test revealed the property advice in the payload of the SafetyNet’s attestation. This field is added to the TLSConnectorServer to be shown as part of the result of the SafetyNet Attestation check. As illustrated in figure 28, SafetyNet’s advice is to lock the device’s bootloader. The bootloader of the Nexus 5X is indeed locked; figure 29 illustrates that the OEM unlocking function itself is disabled with the message “Bootloader is already unlocked”. Locking the bootloader of a device will, as described in section 8.2.2, initiate a factory reset. However, it is not allowed to tamper with Northwave’s Nexus 5X. Therefore, this problem could not be resolved.

Northwave’s Nexus 5X also runs on a custom ROM called LineageOS. This custom ROM, LineageOS version 15.0, is based on Android version 8.0. LineageOS is an open-source operating system that is based on Google’s original Android release; LineageOS focusses on individuality, security, longevity which includes binaries to easily root an Android device [140]. Since LineageOS is a custom ROM, this might also trigger the device integrity check of the SafetyNet Attestation service.

However, it seems that the SafetyNet Attestation check was not able to detect that the Nexus 5X is rooted; SafetyNet’s property `basicIntegrity` was not triggered. Northwave’s penetration testers were questioned about which root implementation is used to root the device, however, nobody seemed to know that. Because of that, the Nexus 5X was connected with a USB cable to a desktop, and the following adb command was executed in the device’s root directory:

```
find . -name "su" 2> /dev/null
```
Bypassing SafetyNet Attestation service

The `adb` command above searches for an object named “su”, which is the root binary, and does not print errors in the console by including “2> /dev/null” in the command. The outcome of this command is illustrated in figure 30. The root binary is located in `/sbin/su`; hence, not in the system partition. Therefore, it is concluded that the Nexus 5X has a systemless root binary installed. Root implementations are described in more detail in section 8.2.4.

```
adb shell:
  $ su
  # find ./name "su" 2> /dev/null
  /sbin/su
```

*Figure 30: Nexus 5X systemless root location*

In order to get a better view of the binaries which are located in the `/sbin` folder on the Nexus 5X, the objects in that folder are listed by using Linux’s `ls` command. The outcome of this command is illustrated in figure 31. In the `sbin` folder, several Magisk files are found. This indicates that the Nexus 5X is probably rooted using the Magisk library.

```
adb shell:
  $ su
  # cd /sbin/
  $ ls
  # bump magisk/world kickstart magiskinit magiskpolicy resetprop su supply
```

*Figure 31: Nexus 5X sbin folder*

8.3.2 Nonrooted experiment

Frida suite is compatible with a nonrooted Android device, this is described in more detail in section 7.2.2. In order to make Frida working on a nonrooted device, it needs to be embedded in the APK of the targeted Android application which then needs to be installed on the device. This section describes an experiment in which the modified APK, with Frida embedded, is tested on the OnePlus 3T. For this experiment, the OnePlus 3T is running on its original Android release without root privileges. Note, that in this experiment the SafetyNet Attestation service is not actively bypassed; a nonrooted device with a certified ROM should always pass the device integrity check. The goal of this experiment is to observe whether the SafetyNet Attestation service can detect a modified APK.

As part of the experiment, the APK of the TLSConnector is modified. This is achieved by following the step-by-step guide described in appendix 15.4 [101]; using this guide, Frida can be injected into an APK. Briefly, the following steps are taken to modify and install the APK:

1. Decompile TLSConnector’s APK.
2. Inject the Frida library into the decompiled application folder.
3. Add a reference to Frida’s library in the decompiled source code of the application.
4. Compile the modified source code, the output of this process is an unsigned APK.
5. Create a certificate that can be used to sign the APK.
6. Sign the APK using the created certificate.
7. Install the signed APK using the `adb` command.

Once the modified TLSConnector’s APK was installed on the OnePlus 3T, it was used to perform the SafetyNet Attestation check. The result of this experiment is...
illustrated in figure 32. The SafetyNet Attestation check failed on SafetyNet’s property `apkCertificateDigestSha256` which is about the integrity of the Android application. In the process of injecting Frida into the TLSConnector, the modified APK was signed with a certificate (step 6). However, this was not with the same certificate as the one which was originally used. It seems that this is the reason why the SafetyNet Attestation check failed.

8.3.3 Rooted experiments
This section discusses several experiments using the two discussed root implementations. Those two root implementations are deployed on the OnePlus 3T to analyze whether the SafetyNet Attestation service can detect it. Root privileges are needed to run Frida’s framework as discussed in section 7.2.1. Since both root implementations will be assessed as part of the experiments, this section is split into the two following topics:

- System root
- Systemless root

In order to root the OnePlus 3T, several steps need to be taken. To support this process, LineageOS’s installation guide was used as preparation for each experiment in this section [141]; this installation guide is applicable for both root implementations. The steps which are taken as part of the experiments are discussed below in more detail. Briefly, the process involves the following steps:

1. Unlock bootloader
2. Flash recovery with TWRP
3. Format data in TWRP
4. Clean the system and cache partitions in TWRP
5. Install Android ROM using TWRP
6. Install root implementation using TWRP

In order to root the OnePlus 3T, the bootloader needs to be unlocked. This enables to reflash partitions on the device; reflashing can be installing an Android ROM or a root binary into the file system. Therefore, the first step was to unlock the bootloader of the OnePlus 3T. The bootloader fastboot command of the OnePlus 3T is unlockable through the user interface; this could be achieved by setting OEM Unlocking on true. This enabled the functionality to unlock the bootloader in fastboot mode. OnePlus 3T’s fastboot mode can be accessed by holding the volume up + power button during the startup of the device until it is booted. Once the OnePlus 3T was in fastboot mode, the bootloader was unlocked with the following fastboot command:

```
fastboot oem unlock
```

For all rooting experiments in this section, the OnePlus 3T’s bootloader state remains unlocked. The reason for this is because it can be dangerous to lock the bootloader after modifications are made to any partition on the device. After the bootloader’s state will be changed to locked, a factory reset will be performed, and Verified Boot will be enforced. When Verified Boot detects a partition that is tampered with, it may not boot the operating system. However, since a factory reset was initiated, the OEM unlocking option will set to false. This means that the bootloader cannot be unlocked in fastboot mode, without interaction in the user interface. Therefore, locking the bootloader after tampering with a partition, can result in a bricked Android device. A bricked Android device means that the operating system cannot be booted.

The unlocked bootloader gives the opportunity to flash partitions like the recovery, boot and system partition. LineageOS’s installation guide suggests to take advantage of this by installing a custom recovery tool called TWRP. TWRP stands for Team Win Recovery Project.
which is an open-source software recovery image for Android-based devices. It provides a touchscreen-enabled interface that allows users to manage the device’s filesystem and to flash partitions; those functions are often not supported by the stock recovery image [142, 143]. Each Android device model type needs its own TWRP release; so far TWRP supports 62 different brands and for each brand one or more models [144]. TWRP also supports the OnePlus 3T [145]. Therefore, TWRP was flashed into the recovery partition of the OnePlus 3T. This was achieved by downloading the latest TWRP and starting the device in fastboot mode. The following fastboot command was executed to flash the partition:
fastboot flash recovery twrp-3.3.1-1-oneplus3.img

After the recovery partition of the OnePlus 3T was flashed with TWRP, it needed to be booted into recovery mode; this needs to be done at startup of the device and the button combination differs for each device. For the OnePlus 3T, this could be done by holding the volume down + power button during the startup of the device until it is booted. Once the device was booted, the TWRP home screen was shown; this TWRP home screen is illustrated in figure 33.

In order to flash an operating system on the device, the partitions need to be formatted. This process removes the encryption and deletes all files stored in the internal storage of the device. Furthermore, the cache and system partition needs to be wiped as well. This removes the operating system from the device. Those two processes could be performed in TWRP under the option Wipe on TWRP’s home screen.

Finally, the OnePlus 3T was ready to install a new operating system and optionally a root binary. This can be achieved by clicking Advanced → ADB Sideload on TWRP’s home screen; ADB sideload is functionality to push and install a ZIP file on the device [146]. Figure 34 illustrates the screen wherein the device is listening to adb commands from the desktop. When the device is in this state, the following adb command can be used to install packages such as an Android ROM or root binary:
adb sideload <package>.zip

The actual installation of the Android ROM and root binary is discussed below as part of the experiments.

8.3.3.1 System root
The introduction of this section describes the preparation for each rooting experiment. With this preparation, the OnePlus 3T is cleaned and ready to install packages. This part of the section focusses on the system root implementation; system root is discussed in more detail in section 8.2.4.1. Based on the experiments in this section, it can be concluded whether the SafetyNet Attestation service can detect the system root implementation. For system root, the following two experiments are performed:
- OnePlus 3T’s Android release with SuperSU system root
- LineageOS release with addonsu system root
**OnePlus 3T's Android release with SuperSU system root**

In section 8.2.4.1, the SuperSU root library is discussed. This root binary can deploy system root into an Android operating system. For this first experiment, OnePlus 3T's Android release was installed with the SuperSU system root binary. Those packages are installed in the following order:

1. Android version 9.0
2. SuperSU version 2.82

The installation OnePlus 3T's Android release and SuperSU resulted in a bricked device. The operating system could not be booted. Luckily, this could be resolved by cleaning the device using the installation preparation and installing OnePlus 3T's Android release without the root binary. Based on this, it is concluded that the SuperSU library is not compatible with this Android version. To get a better view of the situation, OnePlus 3T's Android release version 8.0 was installed together with the SuperSU library. Unfortunately, the result was the same as with Android version 9.0, the device became bricked. Therefore, it is concluded that the SuperSU might not be compatible with OnePlus 3T's Android releases.

**LineageOS release with addonsu system root**

For the second experiment, LineageOS version 16.0 was installed on the OnePlus 3T. LineageOS version 16.0 is based on Android version 9.0 which is the same version as OnePlus 3T's release [147]. LineageOS also distributes a system root binary, called addonsu, which can be used to root an instance of LineageOS [148]. Furthermore, LineageOS does by default not include the Google Play services in its release; however, it does still support it by distributing it in a separate package [141]. Google Play services is required to support the SafetyNet Attestation service. LineageOS's Google play services distribution which is used for this experiment is called MindTheGapps [149]. In order to install LineageOS with system root on the OnePlus 3T, the packages are installed in the following order:

1. LineageOS version 16.0
2. MindTheGapps version 9.0
3. Addonsu version 16.0

Once LineageOS was installed on the OnePlus 3T, the TLSConnector was installed to analyze whether the system root implementation could be detected. Figure 35 illustrates the outcome of this experiment; the SafetyNet Attestation check failed on SafetyNet's properties ctsProfileMatch and basicIntegrity. SafetyNet also provides an advice with the values RESTORE_TO_FACTORY_ROM and LOCK_BOOTLOADER. Also, the OEM unlocking functionality is disabled; this is the same as for Northwave's Nexus 5X which is illustrated in figure 29.

Since the SafetyNet Attestation check failed on both device integrity properties, it seems that the system root was detected. This is even more clear by SafetyNet's advice value RESTORE_TO_FACTORY_ROM, which advises to restore the device to its factory ROM.
Furthermore, figure 36 illustrates the location of the root binaries which are located in the system partition; this command is executed in the device’s UNIX shell using the adb tool.

8.3.3.2 Systemless root
The introduction of this section describes the preparation for each rooting experiment. With this preparation, the OnePlus 3T is cleaned and ready to install packages. This part of the section focuses on the systemless root implementation; systemless root is discussed in more detail in section 8.2.4.2. Based on the experiments in this section, it can be concluded whether the SafetyNet Attestation service can detect the systemless root implementation.

In section 8.2.4.2, the Magisk library is discussed which can be used to deploy systemless root in an Android operating system. According to several sources, Magisk can go undetected by the SafetyNet Attestation service; the system partition remains unchanged [150, 151]. This section covers the following two experiments regarding systemless root:

- OnePlus 3T’s Android release with Magisk systemless root
- LineageOS release with Magisk systemless root

OnePlus 3T’s Android release with Magisk systemless root
For this first experiment, OnePlus 3T’s Android release was installed with the Magisk library. Those packages are installed in the following order:

1. Android version 9.0
2. Magisk version 20.3

Once OnePlus 3T’s Android release was deployed, the TLSConnector was installed to analyze whether the systemless root implementation could be detected. Figure 37 illustrates the outcome of this experiment; the SafetyNet Attestation check passed. It seems that the SafetyNet Attestation service was not able to detect the systemless root binary. Also, the bootloader of the OnePlus 3T remained open after the installation of the Android ROM; the SafetyNet Attestation service was not able to detect this. Furthermore, figure 38 illustrates the location of the root binary which is located outside the system partition; this command is executed in the device’s UNIX shell using the adb tool.

In section 8.2.4.2, the Magisk library is discussed which can be used to deploy systemless root in an Android operating system. According to several sources, Magisk can go undetected by the SafetyNet Attestation service; the system partition remains unchanged [150, 151]. This section covers the following two experiments regarding systemless root:

- OnePlus 3T’s Android release with Magisk systemless root
- LineageOS release with Magisk systemless root

OnePlus 3T’s Android release with Magisk systemless root
For this first experiment, OnePlus 3T’s Android release was installed with the Magisk library. Those packages are installed in the following order:

1. Android version 9.0
2. Magisk version 20.3

Once OnePlus 3T’s Android release was deployed, the TLSConnector was installed to analyze whether the systemless root implementation could be detected. Figure 37 illustrates the outcome of this experiment; the SafetyNet Attestation check passed. It seems that the SafetyNet Attestation service was not able to detect the systemless root binary. Also, the bootloader of the OnePlus 3T remained open after the installation of the Android ROM; the SafetyNet Attestation service was not able to detect this. Furthermore, figure 38 illustrates the location of the root binary which is located outside the system partition; this command is executed in the device’s UNIX shell using the adb tool.
LineageOS release with Magisk systemless root

For the second experiment, LineageOS version 16.0 was installed on the OnePlus 3T [147]; this is the same version that was installed for the system root experiment in section 8.3.3.1. LineageOS does by default not include the Google Play services in its release; however, it does still support it by distributing it in a separate package [141]. Google Play services is required to support the SafetyNet Attestation service. LineageOS's Google play services distribution which is used for this experiment is called MindTheGapps [149]. In order to install LineageOS with systemless root on the OnePlus 3T, the packages are installed in the following order:

1. LineageOS version 16.0
2. MindTheGapps version 9.0
3. Magisk version 20.3

Once LineageOS was deployed on the OnePlus 3T, the TLSConnector was installed to analyze whether the systemless root implementation could be detected. Figure 39 illustrates the outcome of this test; the SafetyNet Attestation check fails on SafetyNet’s property ctsProfileMatch. This is the same property as Northwave’s Nexus 5X, described in section 8.3.1, is failing on. Like in the previous experiment, the bootloader of the OnePlus 3T is still open. However, SafetyNet did not give an advice for that; this is different compared to Northwave’s Nexus 5X. Therefore, it seems that the SafetyNet Attestation check was only able to detect that LineageOS is a custom ROM. Furthermore, figure 38 illustrates the location of the root binaries which are located outside the system partition; this command is executed in the device’s UNIX shell using the adb tool.

Unlocked bootloader

Interestingly, it seems that the SafetyNet Attestation service was not able to detect the open bootloader in both systemless root experiments. After doing some more research about this, two articles were found about hiding root and passing the SafetyNet Attestation service with Magisk. In those articles, it is stated that in March 2020 Google updated the SafetyNet Attestation service to detect the unlocked bootloader state. Before this moment, Magisk was able to spoof the bootloader’s state for any device.

However, Google’s update only affects Android devices which have a hardware-backed key attestation for the bootloader’s state; this hardware-backed mechanism is also part of Verified Boot which is discussed in detail in section 8.2.3. Any device shipped with Android version 8.0, is required to have such a hardware-backed mechanism [152, 153]. Furthermore, on the Android developer’s website, the following statement is posted [123]:

Figure 39: LineageOS systemless root experiment SafetyNet Attestation check
“Most likely, the device launched with an Android version less than 7.0 and it does not support hardware attestation. In this case, Android has a software implementation of attestation which produces the same sort of attestation certificate, but signed with a key hardcoded in Android source code. Because this signing key is not a secret, the attestation could have been created by an attacker pretending to provide secure hardware.”

The OnePlus 3T was originally shipped with Android version 6.0.1, therefore, this was the first indication that the device does not have a hardware-backed mechanism [154]. In order to confirm that, an Android application called “Key Attestation Demo” was downloaded from the Google Play store [155]. The goal for this app is to test the device’s hardware-backed key attestation. Figure 40 illustrates the home screen of the Key Attestation Demo in which it is stated that the OnePlus 3T does not have the support for it. Based on that the OnePlus 3T’s bootloader’s state could be hidden, the OnePlus 3T was released with Android version 6.0.1 and the result in the Key Attestation Demo application; it can be concluded that the OnePlus 3T does not support hardware-backed key attestation.

Figure 40: OnePlus 3T output Key Attestation Demo
9 Bypassing certificate pinning and SafetyNet Attestation service

This chapter describes the final result of this research project in which the results from chapters 7 and 8 are combined and tested together. Hence, the Android-CertificatePinning-Killer and the result of bypassing the SafetyNet Attestation service are used together as the final result. In section 9.1, this final result is tested against the TLSConnector. Finally, section 9.2, is dedicated to testing the final result against several Android applications from the Google Play store.

9.1 Introduction

The Android-CertificatePinning-Killer can bypass certificate pinning for Android’s default library and the OKHttp library. It also includes a logging mechanism for when the SafetyNet Attestation API is called by the Android application, and the result of SafetyNet’s evaluation which is returned by Google’s server. In order to bypass the SafetyNet Attestation service, the OnePlus 3T Android device is deployed with its stock ROM, which is Android version 9.0, and systemless root using Magisk version 20.3.

In order to test the final result against the TLSConnector, the Android-CertificatePinning-Killer is deployed on the OnePlus 3T. Furthermore, to deploy a man-in-the-middle position, Burp Suite is deployed as well. Figure 41 illustrates the outcome of this test in which the response sent from the TLSConectorServer is eavesdropped and shown in plain text. This response comprises the following message: “Android SafetyNet Attestation check passed”. Herby, it can be concluded that the test was successful; the protection of the data sent in a TLS session, with certificate pinning and SafetyNet Attestation service implemented, can be broken in an Android environment.

Figure 41: Android-CertificatePinning-Killer check response in Burp Suite
9.2 Testing in the wild

In order to test the final result against several Android applications from the Google Play store, the same setup is used as described in section 9.1. The Android-CertificatePinning-Killer is deployed on the OnePlus 3T with systemless root. With this setup, the SafetyNet Attestation check will pass. Furthermore, by deploying the Android-CertificatePinning-Killer, it can be determined whether the targeted application is using the SafetyNet Attestation service. When the functions of the SafetyNet Attestation API are successfully hooked, the application is using the service.

In order to find Android applications that are candidates for this test, forums are observed to search for Android applications that are using the SafetyNet Attestation service [156, 157]. Surprisingly, the SafetyNet Attestation service is not very popular. The release date of the SafetyNet Attestation service is not stated anywhere, however, the first sample code of this service appeared on GitHub in August 2016 [158]. In August 2017, a company called NowSecure published an article wherein the author downloaded 3000 popular Android applications from the Google Play store. Those applications were used to analyze to determine which of them were using the SafetyNet Attestation service. The conclusion was that only 0.77% of the applications were using it [159]. Furthermore, in February 2020, a company called NetGuru published an article about the SafetyNet Attestation service in which it also stated that the service is not popular [160]. In both articles, the authors did not know why Android developers are not taking advantage of the SafetyNet Attestation service.

Furthermore, for the tests in this section, Frida is used to target applications’ processes to inject the Android-CertificatePinning-Killer. To target an Android application, the package name of the application must be known; this is described in detail in section 7.2. In order to find the package name for each Android application, an Android application called the “Apk Analyzer” is used [161]. The goal of the Apk Analyzer is to illustrate details about the infrastructure of the applications which are installed on the Android device; this includes the application’s package name.

Using the information provided on the forums and the Apk Analyzer, the Android-CertificatePinning-Killer was tested on several Android applications. One of the Android applications, which was being tested, is the game called Mario Run. This game is developed by Nintendo. During the test, it was also observed that the other games developed by Nintendo are using the SafetyNet Attestation service; those games are also included in the test. Furthermore, to get a better view of whether popular banks and government applications in the Netherlands are using the SafetyNet Attestation service, those are also included in the test as well.

In total 17 Android applications were tested; the result of this test is illustrated in appendix 15.5. In 11 out of the 17 tested Android applications, the data sent in a TLS session could be observed in Burp Suite. Surprisingly, none of the Dutch banking or government applications were using the SafetyNet Attestation service. This was not expected, the SafetyNet Attestation service could also be used to support the risk management of an Android application. For instance, it can be used to offer users only part of the application functionality when a device is rooted.
10 Conclusion

This chapter answers the research question of this study:

“Can the protection of the data sent in a TLS session, with certificate pinning and SafetyNet Attestation service implemented, be broken in an Android environment?”

In short, the answer to the research question is; yes, the protection of the data sent in a TLS session, with certificate pinning and SafetyNet service implemented, can be broken in an Android environment. In order to obtain the required information to answer the research question, sub-research questions were used and are answered below. The answer to the 8th sub-research question gives a more detailed explanation about how the protection of a TLS session can be broken in an Android environment.

1. How does the TLS protocol achieve confidentiality, data integrity and authenticity of the information exchanged between two communicating parties?

The TLS protocol uses a combination of asymmetric and symmetric cryptography to achieve confidentiality. Whether confidentiality is achieved, depends on the strength of the session key. To achieve data integrity, messages in a TLS session contains a Keyed-Hashing Message Authentication Code (HMAC). HMAC uses a cryptographic hash function in combination with a session key. The strength of the HMAC depends on the underlying hash function. Authenticity is achieved during the TLS Handshake protocol. In this protocol, the digital certificate of the server, and optionally the client, are transmitted. A digital certificate can be used to verify the identity of the other party. The communicating parties are responsible to verify the validity of the provided certificate. The answer to this research question is explained in more detail in section 2.4.

2. What are the known weaknesses of the TLS protocol?

In this study, several weaknesses are discussed of the TLS protocol. The weaknesses were categorized in the following topics:

- Cryptographic
- Version
- Implementation

The weaknesses described in this study are found over the years for the SSL and TLS protocols. However, a lot of these weaknesses are patched in newer versions of the SSL and TLS protocols. There are too many weaknesses of the TLS protocol to point out here. However, most weaknesses can be fixed by simply only allowing the newest TLS protocol version, to build up a TLS session. At the time of writing (May 2020), the newest TLS protocol version is 1.3. The answer to this research question is explained in more detail in section 2.5.

3. How does certificate pinning work?

Certificate pinning is a certificate validation technique that associates a host with a specific certificate in the chain of trust. The certificate pinning technique needs to be implemented in the application logic as part of the TLS Handshake protocol. Certificate pinning can prevent man-in-the-middle attacks for the TLS protocol. With this technique, it is harder for an attacker to inject its own certificate during the TLS handshake. The answer to this research question is explained in more detail in section 3.4.3.
4. How can the TLS protocol be implemented in an Android environment?

The Android operating system is shipped with a set of trusted CA certificates which can be used to authenticate a server for a TLS session. As of Android version 4.2, Android contains over 100 well-known trusted CAs which are updated in each release; those trusted CAs are stored in the device’s Keystore. The TLS protocol can be implemented in an Android environment by using Java’s default java.net library. This library handles the authentication of the communicating party by using the device’s Keystore. The answer to this research question is explained in more detail in section 4.2.

5. How can certificate pinning be implemented in an Android environment?

In an Android environment, the HttpsURLConnection class can be used to configure certificate pinning. In this study, this approach is called Android’s default library. Android’s default library only uses functions that are part of the Android operating system. By using Android’s default library, the device’s Keystore will not be used to authenticate certificates. Instead, it uses a manually created Keystore which contains the pinned certificate(s).

In this study another popular approach is discussed, this is the OKHttp library. OKHttp is a third-party library that supports modern TLS features such as certificate pinning. The implementation of the OKHttp library uses the built-in TLS implementation to set up a TLS connection. In comparison with Android’s default library, the OKHttp library pins a certificate by a hard-coded, as a string, fingerprint of the certificate; Android’s default library does this by pinning the certificate as a whole.

There may be various other libraries to implement certificate pinning in an Android environment. However, according to Northwave, Android’s default library and the OKHttp library are the most commonly used libraries by their customers to implement certificate pinning. The answer to this research question is explained in more detail in section 4.3.

6. What are the security checks the SafetyNet Attestation service is performing?

The SafetyNet Attestation service is an anti-abuse service that allows, when implemented in the Android application and its server, to assess the Android device that the Android application is running on. Furthermore, this service can determine whether the application’s backend server is interacting with the corresponding genuine Android application on a genuine Android device.

The SafetyNet Attestation service is focused on the integrity of the Android application and the Android device which it is running on. This can be determined by an assessment that is performed by the SafetyNet Attestation service. This assessment contains several checks in which the outcome is sent to the application server. Based on the outcome of the assessment, the application server can make anti-abuse decisions. This means that the server decides what is considered to be acceptable.

In order to support the integrity of the Android application, the following checks are performed on the application:
1. APK package name
2. Certificate which signed the APK installed on the device
In order to support the integrity of the Android device, which is running the application, the following checks are performed to determine whether the device:

1. Is certified
2. Has an unlocked bootloader
3. Is genuine but not certified
4. Is running a custom ROM
5. Is an Android emulator
6. Is an Android device
7. Has signs of system integrity compromise (for example rooted)
8. Has signs of an active attack

The list describing the integrity of the Android device might not be complete as it is based on the SafetyNet Attestation’s documentation which illustrates these scenarios as examples. Furthermore, based on this list, there are two integrity verdicts; ctsProfileMatch and basicIntegrity. The field ctsProfileMatch is true when only the 1st scenario is satisfied; when one or more of the scenarios of 2-8 is satisfied, the value of ctsProfileMatch is false. The field basicIntegrity is more lenient in which its value is true; except when one or more of the scenarios of 5-8 is satisfied, in that case, the value of basicIntegrity is false. The answer to this research question is explained in more detail in section 5.2.

7. How has the SafetyNet Attestation service patched the Android-SSL-Trustkiller?

The Android-SSL-Trustkiller is a tool to bypass certificate pinning in an Android environment. Using this tool, it was possible to deploy a man-in-the-middle position. The Android-SSL-Trustkiller is using the Cydia Substrate framework. The Cydia Substrate framework works for Android versions 2.3 through 4.3 and requires the device to be rooted.

The Android-SSL-Trustkiller disables the certificate pinning technique. Because of this, the device’s Keystore is used for certificate validation. Therefore, the CA certificate, used to deploy a man-in-the-middle attack, needs to be added to this Keystore.

The SafetyNet Attestation service patched the Android-SSL-Trustkiller because it can detect rooted Android devices. This problem can be solved when the Android device uses systemless root; this is explained in more detail in the answer to the 8th sub-research question. Furthermore, the Cydia Substrate framework is not supported for Android versions above 4.3. Moreover, for Android versions above 6.0, user-added CA certificates to the Keystore, are not considered as trusted anymore. Hence, the Cydia Substrate framework only works for older Android versions. The answer to this research question is explained in more detail in section 7.1.

8. Can certificate pinning and the SafetyNet Attestation service be bypassed in order to break the TLS protection?

Based on the experiments performed in this research project, it can be concluded that the protection of the data sent in a TLS session can be broken. The experiments were tested on several Android applications, with certificate pinning and SafetyNet service implemented, in an Android environment. This thesis proposed a solution to bypass certificate pinning for an Android application with the SafetyNet Attestation service implemented. This solution is applicable for the following certificate pinning libraries:

- Android’s default library
- OKHttp library
The solution is similar to the Android-SSL-Trustkiller which is using the Cydia Substrate framework to deploy code modification at run-time. The proposed solution for this research project is a code modification script, called the Android-CertificatePinning-Killer, which requires the deployment of the Frida framework. The Frida framework can deploy, like the Cydia Substrate framework, code modification at run-time by targeting an Android application. To make the Frida framework working, it needs to be deployed on an Android device and desktop; it is using the client-server principle in which the Android device is the server. Frida works on both nonrooted and rooted Android devices. However, to bypass the SafetyNet Attestation service, a rooted Android device is required.

In order to bypass the SafetyNet Attestation service on a rooted Android device, the Android device needs to be deployed with systemless root. Systemless root does not tamper with the system partition. Based on the experiments performed for this study, it seems that the SafetyNet Attestation check fails when the system partition is tampered with. Tampering with the system partition happens when an Android device is deployed with system root. For this research project, the SafetyNet Attestation service can be bypassed with a OnePlus 3T Android device. To bypass the service, it needs to be deployed with its stock Android ROM version 9.0 and Magisk version 20.3 systemless root library.

In order to deploy systemless root, the bootloader remained open. However, in March 2020 Google updated the SafetyNet Attestation service to detect the unlocked bootloader state with hardware-backed key attestation. Before this moment, Magisk was able to spoof the bootloader’s state for any device. The OnePlus 3T does not have such hardware-backed key attestation. Therefore, it is assumed that bypassing the SafetyNet Attestation service only works on Android devices without support for hardware-backed key attestation. The answer to this research question is explained in more detail in sections 7.3, 8.3 and 9.1.
11 Discussion and future work

The proposed solution to bypass certificate pinning for an Android application is tested on the following Android devices:

- Nexus 5X running LineageOS version 15.0 (based on Android 8.0) with root privileges
- OnePlus 3T running its stock Android version 9.0 with root privileges
- Xiaomi Mi 9T running its stock Android version 9.0 without root privileges

In order to bypass Android's default library, the Android-CertificatePinning-Killer hooks functions which are part of the Android operating system. Since those functions are part of the operating system, it is assumed that the solution works on at least Android versions 8.0 and 9.0. To bypass the OKHttp library, the Android-CertificatePinning-Killer hooks functions of the library itself. It also uses a part of the solution to bypass Android's default library. Therefore, the compatibility to bypass the OKHttp library might depend on which version of the OKHttp library the targeted Android application is using. The TLSConnector, on which the experiments are performed, is using OKHttp version 4.2.1. When an older or newer version of the OKHttp library is used by the targeted Android application, the solution might not work depending on whether the function which the script is hooking, is still in the same package and has the same parameters. For future research, it might be interesting to test the Android-CertificatePinning-Killer on more Android devices and Android versions.

In order to bypass the SafetyNet Attestation service, it is studied in detail. However, in the service's documentation, it is not explained how the SafetyNet Attestation check is performed. Due to lack of time, in this study were no experiments performed to actively bypass the SafetyNet Attestation check. Therefore, investigating how the SafetyNet Attestation check is performed and to actively bypassing it, is left for future work.

In the nonrooted experiment to bypass the SafetyNet Attestation service, the APK of the targeted Android application was modified to inject the Frida Gadget in the application's source code. An alternative experiment that could also disable certificate pinning, is to simply remove the certificate pinning technique from the application's source code. However, as with the nonrooted experiment, this will trigger the SafetyNet Attestation check; it may trigger SafetyNet's property apkCertificateDigestSha256. This is caused by that the APK is signed with a different certificate during the modification process. When the targeted Android application is not using the SafetyNet Attestation service, this approach might be interesting to further investigate.

In order to bypass the SafetyNet Attestation service, several rooting experiments were performed. With those rooting experiments, system and systemless root implementations were assessed on a OnePlus 3T Android device. Unfortunately, there were no other Android devices available for the experiments. Since the experiments were only tested on one Android device, representativeness on other Android devices can be improved. Therefore, it would be interesting for a future study to test the experiments on several other Android devices and Android versions.

For the systemless root experiments, the Magisk library was deployed on a OnePlus 3T Android device. By deploying the OnePlus 3T with its stock ROM and Magisk systemless root, the SafetyNet Attestation service could be bypassed. Furthermore, by using the Magisk library, the unlocked bootloader could not be detected. However, since March 2020, Google updated the SafetyNet Attestation service to detect the unlocked bootloader state. With this update, it will check the bootloader state by using hardware-backed key attestation. This check
cannot be bypassed (yet). However, it only affects Android devices with this hardware-backed mechanism. The hardware-backed key attestation mechanism is enforced for Android devices that are shipped with Android versions 8.0 or above. For this study, there was no Android device with such a hardware-backed mechanism available. Because of this, Google’s update could not be tested as part of the rooting experiments. Therefore, based on this research, it can be concluded that bypassing the SafetyNet Attestation service works on Android devices without a hardware-backed key attestation mechanism. No further conclusions can be made for Android devices with support for such hardware-backed mechanism. For a future study, it might be interesting to do an in-depth analysis of the hardware-backed key attestation. By finding exploits in this mechanism, a new way to bypass the bootloader lock state may be found for newer Android devices.

This research project was focused on breaking the protection of TLS sessions in an Android environment with the SafetyNet Attestation service implemented. To bypass the SafetyNet Attestation service, the rooting process of Android devices was studied in detail. Android devices are Linux-based and by rooting an Android device, root access can be gained. With this root access, which is also known as superuser, the user can manage the Android system with administrative privileges. This also means that the user can access any object in the device’s filesystem. Therefore, root privileges can also be used to analyze locally stored (user) data which is being processed by an Android application. Therefore, it may be interesting to use the root privileges to assess what is being stored by the Android application as part of the penetration test process.

At the end of this research project, the Android-CertificatePinning-Killer was tested on several Android applications. This test was performed on a rooted OnePlus 3T Android device which passes the SafetyNet Attestation check. To test the Android-CertificatePinning-Killer, Android applications that are using the SafetyNet Attestation service were being searched. Surprisingly, the SafetyNet Attestation service seemed not to be very popular. Moreover, several Dutch banking and government applications were tested; none of them seemed to use the SafetyNet Attestation service either. No clarification could be found about why the SafetyNet Attestation service is unpopular. It may be interesting for a future study to find out why that is the case.

The Android-CertificatePinning-Killer did not work on all tested Android applications; the traffic could not be observed in all tests. There could be many reasons why it was not working. One of the reasons why the Android-CertificatePinning-Killer might not work, is that the targeted application is using root detection which detects the systemless root implementation. Another clarification could be that the targeted Android application is performing a manual check on the TLS connection; those checks are not hooked by the Android-CertificatePinning-Killer. Due to a limited time frame, clarification of why the Android-CertificatePinning-Killer is not working on all tested Android applications is left for future work. Note that when the Android application is using root detection or is performing other manual checks, the functions which are responsible for this could be hooked with the Frida framework.
12 Acknowledgment

This thesis is written as part of the master project that I have performed at Northwave, Utrecht, the Netherlands. The master project was performed for my master’s in Computing Science with a specialization in Cyber Security at Radboud University, Nijmegen, the Netherlands. The research project was started on the 1st of November 2019 and finished on the 11th of May 2020.

I would first like to thank my thesis supervisor Prof. dr. Eric Verheul of Faculty of Science at Radboud University. During the course of this research project, I have had biweekly meetings with him wherein we discussed the progress of the project. Those meetings helped me a lot with steering me in the right direction whenever he thought I needed it.

Secondly, I express my gratitude to Northwave for giving me such an exciting topic for my master’s thesis. I have gained interesting knowledge about how the Android environment works and how security protocols can be implemented in it. Also, I would like to thank Frank de Korte which was my supervisor at Northwave. By having a biweekly meeting with him, he gave me interesting insights to tackle the research question of this thesis. This helped me a lot and I was eventually able to answer the question.

Also, a big thanks goes to my colleagues in Northwave’s Red team for helping along the way. This team helped me a lot by answering the security questions I had related to this research project. Finally, I would like to thank my parents for their great support throughout my study years at Radboud University.

I have enjoyed working on this research project, mainly because of the freedom I had for a practical approach to tackle the research question. It was very exciting to use the knowledge of the literature review, described in chapters 2,3,4 and 5, to set up a test environment that is described in chapter 6. For this test environment an Android application, called the TLSConnector, was developed along with its backend server. By setting up this test environment, it was very interesting to see how security controls, such as certificate pinning and the SafetyNet Attestation service, could be implemented in an Android application. In addition to that, it was nice to see that this test environment could be used to perform experiments and eventually break certificate pinning in an Android environment with the SafetyNet Attestation service implemented.
## 13 Terms

<table>
<thead>
<tr>
<th>Term</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adb</td>
<td>Android Debug Bridge (adb) is a client-server tool that can be used to enter the UNIX shell of an Android device from the desktop.</td>
</tr>
<tr>
<td>Android ROM</td>
<td>Android ROM is the firmware which is the Android operating system’s installation file.</td>
</tr>
<tr>
<td>Android-CertificatePinning-Killer</td>
<td>Android-CertificatePinning-Killer is a tool that disables certificate pinning in an Android environment. This tool is developed as part of the solution for this thesis.</td>
</tr>
<tr>
<td>Android-SSL-Trustkiller</td>
<td>Android-SSL-Trustkiller is a tool that disables certificate pinning in an Android environment. This was the old solution for Northwave to bypass certificate pinning.</td>
</tr>
<tr>
<td>Android’s default library</td>
<td>Android’s default library is a certificate pinning library which uses functions of the operating system.</td>
</tr>
<tr>
<td>APK</td>
<td>Android application package (APK) is an archive file that contains all the contents of an Android application. With this file, an Android application can be installed on an Android device.</td>
</tr>
<tr>
<td>Bootloader</td>
<td>Bootloader is a vendor-proprietary image that is responsible for bringing up the kernel on a device. When the bootloader’s state is unlocked, partitions on the Android device can be flashed.</td>
</tr>
<tr>
<td>Burp Suite</td>
<td>Burp Suite is an application that allows to set up a web proxy that can be used to intercept, inspect and modify raw traffic passing in both directions. Burp Suite uses its own CA certificate which signs other certificates, generated on the flow, for its proxy connections.</td>
</tr>
<tr>
<td>Certificate authority</td>
<td>Certificate authority (CA) is an organization that issues digital certificates. A CA acts as a trusted third party that signs the issued certificates. When an entity trusts a CA, then the certificates which are signed by that CA can be considered as trusted as well.</td>
</tr>
<tr>
<td>Certificate chain</td>
<td>Certificate chain refers to that digital certificates can be signed hierarchically. This means that certificates can be signed by a trusted party (CA) and that those parties use their certificate, to sign other certificates. This process chains the certificates.</td>
</tr>
<tr>
<td>Certificate pinning</td>
<td>Certificate pinning is a certificate validation technique that associates a host with a specific certificate in the certificate chain.</td>
</tr>
<tr>
<td>Certificate validation</td>
<td>Certificate validation is the process to verify the validness of a digital certificate. This can be done by calculating the chain of trust or using techniques such as certificate pinning.</td>
</tr>
<tr>
<td>Chain of trust</td>
<td>Chain of trust refers to how an entity’s certificate can be linked back to a trusted party in the certificate chain.</td>
</tr>
<tr>
<td>Cydia Substrate framework</td>
<td>Cydia Substrate framework can deploy code modification at run-time by hooking function calls. This framework is used in the Android-SSL-Trustkiller.</td>
</tr>
<tr>
<td>Terms</td>
<td>Definition</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Digital certificate</td>
<td>Digital certificate, in this thesis called a certificate, is an electronic document used in the TLS protocol to prove the ownership of a public key. When the certificate is signed by a trusted party (CA), it can also be used as proof of identity.</td>
</tr>
<tr>
<td>Fastboot</td>
<td>Fastboot is a client-server tool which can be used for bootloader management and flashing partitions of an Android device from the desktop.</td>
</tr>
<tr>
<td>Flashing</td>
<td>Flashing is the process of installing (custom) software on the device’s partition. An example is flashing an Android ROM which installs the Android operating system on the targeted device.</td>
</tr>
<tr>
<td>Frida framework</td>
<td>Frida framework can deploy code modification at run-time by hooking function calls. This framework is used for the Android-CertificatePinning-Killer.</td>
</tr>
<tr>
<td>Hooking</td>
<td>Hooking is an approach to intercept function calls, messages or events passing between software components. By doing this, the behavior of the targeted software can be altered.</td>
</tr>
<tr>
<td>Keystore</td>
<td>Keystore is a set of trusted certificates in the Android operating system which can be used to authenticate a server for a TLS connection.</td>
</tr>
<tr>
<td>OkHttp library</td>
<td>OKHttp library is a third-party certificate pinning library that uses its own implementation for certificate pinning.</td>
</tr>
<tr>
<td>Rooted android device</td>
<td>Android devices are Linux-based and do by default not have the superuser binary in the filesystem. Rooting an Android device adds this binary so that the user can perform actions with root (superuser) privileges.</td>
</tr>
<tr>
<td>SafetyNet</td>
<td>SafetyNet is a set of services and APIs that helps to protect an Android application against security threats, including device tampering, bad URLs, potentially harmful apps, and fake users.</td>
</tr>
<tr>
<td>SafetyNet Attestation service</td>
<td>SafetyNet Attestation service is an anti-abuse service that allows, when implemented in the Android application and its server, to assess the Android device that the Android application is running on; this includes root detection.</td>
</tr>
<tr>
<td>SSL protocol</td>
<td>Secure Sockets Layer (SSL) protocol is the predecessor of the TLS protocol.</td>
</tr>
<tr>
<td>System root</td>
<td>System root is a root implementation that can be used to root an Android device. With system root, the root (superuser) binary is located in the system partition of the device.</td>
</tr>
<tr>
<td>Systemless root</td>
<td>Systemless root is a root implementation that can be used to root an Android device. With systemless root, the root (superuser) binary is located outside the system partition of the device.</td>
</tr>
<tr>
<td>TLS protocol</td>
<td>Transport Layer Security (TLS) protocol is an application layer protocol designed to establish an encrypted channel between two or more parties.</td>
</tr>
<tr>
<td>TWRP</td>
<td>TWRP (Team Win Recovery Project) is an open-source software recovery image for Android-based devices. It provides a touchscreen-enabled interface that allows users to manage the device’s filesystem and to flash partitions.</td>
</tr>
</tbody>
</table>
14 References


References


References


Appendix

15 Appendix

This chapter describes additional information for this thesis. In several sections, a desktop is involved. Since that Northwave main works with the macOS environment, the sections are written to be performed on this environment.

15.1 Setup Burp Suite for a man-in-the-middle position

This section describes a step-by-step guide of how a man-in-the-middle position can be achieved in an Android environment using the desktop application called Burp Suite. For this guide, the Android device needs to be on the same network as the desktop. Also, the local IP address of the desktop needs to be known.

Desktop requirement:
- Burp Suite Community Edition

1. Start a proxy using Burp Suite

Start Burp Suite and go to Proxy -> Options -> “Proxy Listeners”. Output:

Press “Add” and bind the proxy to port “8080” and select “All interfaces”. Output:

https://portswigger.net/burp
Go to the tab “Request handling” and select “Support invisible proxying”. Output:

Press “OK” and confirm. Output:

2. Turn off Burp’s intercept functionality
Turn off the intercept functionality of Burp Suite by going to Proxy -> Intercept and press “Intercept is off”. Output:
3. Setup proxy on Android device
Connect the Android device to the same network as the desktop. On the device, go to Settings -> Wi-Fi -> Modify Network -> “Proxy” and set this setting to “Manual” and set desktop’s local IP address and port “8080”. Output:

Note that the exact steps may not be the same for every Android device. However, the goal is to setup a proxy on the Android device to the desktop’s IP address and port “8080”.

15.2 Setup Frida inject code modification script
This section describes a step-by-step guide about how to inject a code modification script on the Android device using Frida 5. In order to do this, the Frida framework needs to be running on the device. On a rooted device, this can be achieved by transferring the Frida-server to the device and execute it; this process is described in section 15.3. On a nonrooted device, this can be achieved by injecting the Frida-gadget into the targeted Android application, install this application; this process is described in section 15.4.

Desktop requirement:
• Frida client

Android device requirement:
• Frida server (rooted device) or Frida Gadget (nonrooted device)

5 https://frida.re/
For this guide the TLSConnector is used and the following code modification script:

```javascript
function() {
    var URL = new URL("https://example.com");
    URL.openConnection().implementation = function() {
        console.log("Connection is opened");
        return this.openConnection();
    }, ());
}
```

This script will print the text “Connection is opened” in the terminal when the `openConnection()` function of the URL class is called.

1. **Search for processes on the device**
   Connect the Android device to the desktop using a USB cable and search for active processes on this device using the following command:
   ```
   frida-ps -U
   ```

   **Output using a rooted device:**

   ```
   PID Name
   3425 ATFMID-reminder
   3429 adbd
   3430 android.hardware.biometrics.fingerprint2.1-service
   414 android.hardware.caha1.0-service
   415 android.hardware.configstore1.0-service
   416 android.hardware.dumpstate1.0-service.bullet
   417 android.hardware.graphics.allocation2.0-service
   418 android.hardware.usb1.0-service
   419 android.hardware.wifi1.0-service
   428 android.nfc.allocation1.0-service
   3347 audioSevr
   3407 cameraSevr
   3427 cmd
   3422 cmss-sevr
   8748 com.android.browser
   5746 com.android.inputmethod.latin
   6534 com.android.nfc
   5959 com.android.phone
   7228 com.android.providers.calendar
   6665 com.android.smssvr
   5758 com.android.systemui
   7547 com.android.mms
   9031 com.android.browser
   9980 com.example.tlsconnector
   ```

   **Output using a nonrooted device:**

   ```
   PID Name
   15739 Gadget
   ```

2. **Inject code modification script**
   Pick a process from the result of step 1 and inject the code modification script using the following command:
   ```
   frida -U -l <myFridaScript>.js <com.example.tlsconnector>
   ```
Output:
```
NM-MBP-Cees-CMA:FridaScripts nw$ frida -U -l myFridaScript.js com.example.tlsconnector

Frida 12.8.6 - A world-class dynamic instrumentation toolkit
Commands:
  help  -> Displays the help system
  object? -> Display information about 'object'
  exit/quit -> Exit

More info at https://www.frida.re/docs/home/

Attaching...

--My Frida script--
[LG Nexus 5X::com.example.tlsconnector] => []
```
Note that on a nonrooted device, only the process “Gadget” is shown, this is the Frida library which is injected into the Android application. Therefore, when using a nonrooted device, the code modification script needs to be injected into this process.

3. Result code modification
As described above, the script for this guide hooks the openConnection() function of the URL class. When this function is called, by making a connection using the TLSConnector application, it results in the following output:

```
[LG Nexus 5X::com.example.tlsconnector] => connection is opened
Connection is opened
```

15.3 Setup Frida on a rooted Android device
This section describes a step-by-step guide of how to deploy the Frida framework on a rooted Android device.

Desktop requirements:
- Android SDK
- Unxz

1. Get device's architecture
Connect the Android device to the desktop using a USB cable and find out what the device’s architecture is using device’s UNIX shell; execute the following command:

```
adb shell getprop ro.product.cpu.abi
```

Output:
```
NM-MBP-Cees-CMA:FRidaRoot nw$ adb shell getprop ro.product.cpu.abi
arm64-v8a
```

2. Download Frida server
Download Frida-server which is matching the device’s architecture and unzip the file using the following command:

```
unxz frida-server-<12.2.26>-android-arm64.xz
```

Output:
```
NM-MBP-Cees-CMA:FRidaRoot nw$ unxz frida-server-12.2.26-android-arm64.xz
```

6 [https://github.com/frida/frida/releases](https://github.com/frida/frida/releases)
3. Copy Frida server to device

Connect the Android device to the desktop using a USB cable and copy the Frida server to the device using the following command:

```
adb push frida-server-<12.2.26>-android-<arm64> /data/local/tmp/
```

Output:

```
WW-MBP-Cees-CMA:FridaRoot nw5 adb push frida-server-12.2.26-android-arm64 /data/local/tmp/
frida-server-12.2.26-android-arm64: 1 file pushed. 22.2 MByte in 1.991s
```

4. Open UNIX shell with root privileges on the device

Enter the device's UNIX shell using the following command:

```
adb shell
```

Output:

```
WW-MBP-Cees-CMA:FridaRoot nw5 adb shell
bullhead:/ $ 
```

Login as root using the following command:

```
su
```

Output:

```
bullhead:/ $ su
bullhead:/ # 
```

5. Set permissions for Frida server

Set execute permission for the Frida-server using the following command:

```
chmod 755 /data/local/tmp/frida-server-<12.2.26>-android-<arm64>
```

Output:

```
bullhead:/ # chmod 755 /data/local/tmp/frida-server-12.2.26-android-arm64
```

6. Run the Frida server

Execute the Frida-server using the following command:

```
/data/local/tmp/frida-server-<12.2.26>-android-<arm64> &
```

Output:

```
bullhead:/ # /data/local/tmp/frida-server-12.2.26-android-arm64 &
[3] 9277
```

The Frida server is running on the device and is ready to make a connection with the Frida client on the desktop. The process to make this connection is explained in appendix 15.2.

15.4 Setup Frida on a nonrooted Android device

This section describes a step-by-step guide of how to inject the Frida framework into an Android application for a nonrooted device.

Desktop requirements:
- APK file of the Android application
- Android SDK
- Zipalign
- Keytool
1. **Decompile the Android application using the apktool**

   Go to the application’s folder and decompile the application using the following command:

   ```
   apktool d <app>.apk -f
   ```

   **Output:**

   ```
   NW-NBP-Cees-OMA:FridaNonRoot mw5 apktool d app.apk
   I: Using Apktool 2.4.1 on app.apk
   I: Loading resource table...
   I: Decoding AndroidManifest.xml with resources...
   I: Loading resource table from file: /Users/mw/library/apktool/framework/1.apk
   I: Regular manifest package...
   I: Decoding file-resources...
   I: Decoding values */* XMLs...
   I: Boksmling classes.dex...
   I: Boksmling classes.dex...
   I: Copying assets and libs...
   I: Copying unknown files...
   I: Copying original files...
   I: Copying META-INF/services directory
   ```

   This creates a new folder with the application’s APK name:

   ```
   NW-NBP-Cees-OMA:FridaNonRoot mw5 ls
   app app.apk
   ```

2. **Get device’s architecture**

   Connect the Android device to the desktop using a USB cable and find out what the device’s architecture is; execute the following command:

   ```
   adb shell getprop ro.product.cpu.abi
   ```

   **Output:**

   ```
   NW-NBP-Cees-OMA:FridaNonRoot mw5 adb shell getprop ro.product.cpu.abi
   arm64-v8a
   ```

3. **Download Frida Gadget**

   Download Frida Gadget which is matching the device’s architecture and unzip the file using the following command:

   ```
   unxz frida-gadget-<12.2.26>-android-arm64.so.xz
   ```

   **Output:**

   ```
   NW-NBP-Cees-OMA:FridaNonRoot mw5 unxz frida-gadget-12.2.26-android-arm64.so.xz
   ```

4. **Copy Frida Gadget to application’s folder**

   Create a folder named “lib” in the application’s folder using the following command:

   ```
   mkdir <app>/lib
   ```

   **Output:**

   ```
   NW-NBP-Cees-OMA:FridaNonRoot mw5 mkdir app/lib
   ```

   Create a folder in the “lib” folder named after the device’s architecture using the following command:

   ```
   mkdir app/lib/<arm64-v8a>
   ```

   **Output:**

   ```
   NW-NBP-Cees-OMA:FridaNonRoot mw5 mkdir app/lib/arm64-v8a
   ```

---

7 [https://github.com/frida/frida/releases](https://github.com/frida/frida/releases)
Appendix

Copy the downloaded Frida Gadget (step 3) into this folder and rename it to 'libfrida-gadget.so' using the following command:

```sh
cp frida-gadget-<12.2.26>-android-arm64.so app/lib/arm64-v8a/libfrida-gadget.so
```

**Output:**

```
MK-MBP-Cees-MA:FridaNonRoot mv$ cp frida-gadget-12.2.26-android-arm64.so app/lib/arm64-v8a/libfrida-gadget.so
```

5. **Make a reference to Frida Gadget at the starting point of the application**

Open the "AndroidManifest.xml" file in the application's folder and look for the first Activity which is being loaded during the startup of the application. This Activity is encapsulated under the application tag:

In this case the starting point of the application is the "MainActivity". Find the corresponding file, with suffix .smali, to this activity using the following command:

```
find . -name "<Activity>.smali"
```

**Output:**

```
MK-MBP-Cees-MA:FridaNonRoot mv$ find . -name "<Activity>.smali"
./app/small_classes2/com/example/tlsconnector/MainActivity.smali
```

Open this file, in this case "MainActivity.smali", and look for the constructor, example:

```java
# direct methods
.method public constructor <init>()V
   .locals 3
   .line 17
   invoke-direct (p8), android/appcompat/appCompatActivity::<init>()V
   .line 20
   const-string v8, "TLS 1.0"
   const-string v1, "TLS 1.1"
   const-string v2, "TLS 1.2"
   filled-new-array (v8, v1, v2), [Ljava/lang/String;
   move-result-object v8
   input-object v8, p8, lcom/example/tlsconnector/MainActivity->tlsVersions:[Ljava/lang/String;
```

Increase the value after .locals by one, and add a new line with the following two commands under this statement:

```java
   const-string vo, "frida-gadget"
   invoke-static {vo}, Ljava/lang/System;->loadLibrary(Ljava/lang/String;)V
```
6. Increase the application’s version
Open the “apktool.yml” file in the application’s folder. Example:

```yaml
org/apache/commons/code/language/bw/sap_exact_hebrew.txt: "8"
org/apache/commons/code/language/bw/sap_exact_italian.txt: "8"
org/apache/commons/code/language/bw/sap_exact_portuguese.txt: "8"
org/apache/commons/code/language/bw/sap_exact_spanish.txt: "8"
org/apache/commons/code/language/bw/sap_hebrew_common.txt: "8"
org/apache/commons/code/language/bw/sap_language.txt: "8"
org/apache/commons/code/language/bw/sap_rules_any.txt: "8"
org/apache/commons/code/language/bw/sap_rules_french.txt: "8"
org/apache/commons/code/language/bw/sap_rules_hebrew.txt: "8"
org/apache/commons/code/language/bw/sap_rules_italian.txt: "8"
org/apache/commons/code/language/bw/sap_rules_portuguese.txt: "8"
org/apache/commons/code/language/bw/sap_rules_spanish.txt: "8"
org/apache/http/version.properties: "8"
org/apache/http/client/version.properties: "8"
```

Increase the value after versionCode by one. Output:

```yaml
org/apache/commons/code/language/bw/sap_exact_hebrew.txt: "8"
org/apache/commons/code/language/bw/sap_exact_italian.txt: "8"
org/apache/commons/code/language/bw/sap_exact_portuguese.txt: "8"
org/apache/commons/code/language/bw/sap_exact_spanish.txt: "8"
org/apache/commons/code/language/bw/sap_hebrew_common.txt: "8"
org/apache/commons/code/language/bw/sap_language.txt: "8"
org/apache/commons/code/language/bw/sap_rules_any.txt: "8"
org/apache/commons/code/language/bw/sap_rules_french.txt: "8"
org/apache/commons/code/language/bw/sap_rules_hebrew.txt: "8"
org/apache/commons/code/language/bw/sap_rules_italian.txt: "8"
org/apache/commons/code/language/bw/sap_rules_portuguese.txt: "8"
org/apache/commons/code/language/bw/sap_rules_spanish.txt: "8"
org/apache/http/version.properties: "8"
org/apache/http/client/version.properties: "8"
```

7. Re-build the application
Rebuild the application by targeting the application’s folder using the following command:

```
apktool b <app>
```
8. Optimize APK with zipalign

Align the apk in the "dist" folder using the following command:

```
zipalign -p 4 <app>/dist/<app>.apk <app>/dist/<app>-aligned.apk
```

Output:

```
This will create a new aligned APK file in the "dist" folder which is located in the application folder:
```

```
NN-MBP-Cees-OMA:FridaNonRoot nw5 ls app/dist/
app.apk
```

```
8. Optimize APK with zipalign
Align the apk in the "dist" folder using the following command:
```

```
This will create a new aligned APK file in the "dist" folder which is located in the application folder:
```

```
NN-MBP-Cees-OMA:FridaNonRoot nw5 ls app/dist/
app-aligned.apk app.apk
```

9. Create a Keystore

Create a keystore and remember its password when configuring, command:

```
keytool -genkey -alias keystore-keystore-keyStore.pfx -storetype PKCS12 -keyalg RSA -validity 365 -keysize 2048
```

Output:

```
Enter keystore password:
Re-enter new password:
What is your first and last name?
[Unknown]: X
What is the name of your organizational unit?
[Unknown]: X
What is the name of your organization?
[Unknown]: X
What is the name of your City or Locality?
[Unknown]: X
What is the name of your State or Province?
[Unknown]: X
What is the two-letter country code for this unit?
[Unknown]: X
Is CN=X, OI=X, O=X, L=X, ST=X, C=X correct?
[no]: yes
Generating 2.048 bit RSA key pair and self-signed certificate (SHA256WithRSA) with a validity of 365 days
```

```
Enter keystore password:
Re-enter new password:
What is your first and last name?
[Unknown]: X
What is the name of your organizational unit?
[Unknown]: X
What is the name of your organization?
[Unknown]: X
What is the name of your City or Locality?
[Unknown]: X
What is the name of your State or Province?
[Unknown]: X
What is the two-letter country code for this unit?
[Unknown]: X
Is CN=X, OI=X, O=X, L=X, ST=X, C=X correct?
[no]: yes
Generating 2.048 bit RSA key pair and self-signed certificate (SHA256WithRSA) with a validity of 365 days
```
10. **Sign the application**
Use the created Keystore (step 9) to sign the aligned APK file, located in the “dist” folder, which is located in the application folder, using the following command:

```
apksigner sign -ks keyStore.pfx <app>/dist/<app>-aligned.apk
```

**Output:**

```
 VN-MBP-Cees-OMA:FridaNonRoot memo apksigner sign --ks keyStore.pfx app/dist/app-aligned.apk
Keystore password for signer #1:
```

11. **Install the application on the device**
Connect the Android device to the desktop using a USB cable and install the signed APK file on this device using the following command:

```
adb install <app>/dist/<app>-aligned.apk
```

**Output:**

```
 VN-MBP-Cees-OMA:FridaNonRoot memo adb install app/dist/app-aligned.apk
Performing Streamed Install
Success
```

Accept the installation on the device while executing the command above:

![Install via USB](image)

12. **Open the application**
Open the installed application on the device, this will result in a white screen:

![White screen](image)

This means that the Frida Gadget is awaiting connection with the Frida client on the desktop. This process is explained in appendix 15.2.
## 15.5 Test results bypassing certificate pinning and the SafetyNet Attestation service

<table>
<thead>
<tr>
<th>Android application</th>
<th>Category</th>
<th>Safety Net Attestation service</th>
<th>Certificate pinning library</th>
<th>Traffic could be observed</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLSConnector</td>
<td>Other</td>
<td>☑️</td>
<td>Android’s default &amp; OKHttp</td>
<td>☑️</td>
<td>Can be used to search for Android applications. However, downloading applications does not work.</td>
</tr>
<tr>
<td>Google Play Store</td>
<td>Store</td>
<td>✗</td>
<td>None</td>
<td>☑️</td>
<td>Connection failed message during authentication.</td>
</tr>
<tr>
<td>Netflix</td>
<td>Entertainment</td>
<td>☑️</td>
<td>Android’s default</td>
<td>☑️</td>
<td></td>
</tr>
<tr>
<td>Pokémon Go</td>
<td>Game</td>
<td>☑️</td>
<td>Android’s default</td>
<td>✗</td>
<td>Connection failed message during authentication.</td>
</tr>
<tr>
<td>Mario Run</td>
<td>Game</td>
<td>☑️</td>
<td>Android’s default</td>
<td>☑️</td>
<td></td>
</tr>
<tr>
<td>Fire Emblem Heroes</td>
<td>Game</td>
<td>☑️</td>
<td>Android’s default</td>
<td>☑️</td>
<td></td>
</tr>
<tr>
<td>Mario Kart</td>
<td>Game</td>
<td>☑️</td>
<td>Android’s default</td>
<td>☑️</td>
<td></td>
</tr>
<tr>
<td>Animal Crossing Pocket Camp</td>
<td>Game</td>
<td>☑️</td>
<td>Android’s default</td>
<td>☑️</td>
<td></td>
</tr>
<tr>
<td>Dr. Mario World</td>
<td>Game</td>
<td>☑️</td>
<td>Android’s default</td>
<td>☑️</td>
<td></td>
</tr>
<tr>
<td>Rabobank</td>
<td>Financial</td>
<td>✗</td>
<td>OKHttp</td>
<td>☑️</td>
<td></td>
</tr>
<tr>
<td>ABN AMRO</td>
<td>Financial</td>
<td>✗</td>
<td>Android’s default</td>
<td>☑️</td>
<td></td>
</tr>
<tr>
<td>SNS</td>
<td>Financial</td>
<td>✗</td>
<td>Unknown</td>
<td>✗</td>
<td>Application accepts the proxy, but connection could not be observed in Burp Suite</td>
</tr>
<tr>
<td>ING</td>
<td>Financial</td>
<td>✗</td>
<td>Android’s default</td>
<td>✗</td>
<td>Connection failed error</td>
</tr>
<tr>
<td>Bunq</td>
<td>Financial</td>
<td>✗</td>
<td>Android’s default</td>
<td>✗</td>
<td>Connection failed error</td>
</tr>
<tr>
<td>PayPal</td>
<td>Financial</td>
<td>✗</td>
<td>Android’s default</td>
<td>✗</td>
<td>Connection failed error</td>
</tr>
<tr>
<td>DigiID</td>
<td>Government</td>
<td>✗</td>
<td>Android’s default</td>
<td>☑️</td>
<td></td>
</tr>
<tr>
<td>Berichtenbox</td>
<td>Government</td>
<td>✗</td>
<td>Android’s default</td>
<td>✗</td>
<td>Connection failed error</td>
</tr>
</tbody>
</table>