École Polytechnique Fédérale de Lausanne

Double Ratchet for Bluetooth security

by Alessandro Bianchi

Semester Project

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Project Advisor

Dr. sc. Daniele Antonioli
Project Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

February 4, 2021

Abstract

Communication through Bluetooth technologies lacks security standards that measure up to modern secure-by-design protocols. This project addresses this issue by designing, implementing, and evaluating the use of the Double Ratchet algorithm at the Application-Layer, on top of a Bluetooth-based application. The practical implementation is then followed by an analysis of architectural vulnerabilities of the Double Ratchet algorithm. Finally, attacks based on these vulnerabilities are designed and tested on commonly used applications.

Contents

Chapter 1

Introduction

The aim of this report is to summarize the results achieved throughout six months of research on how to improve Bluetooth security using the Double Ratchet algorithm.

The first section of the report serves the purpose of spelling out the motivation behind this research, by briefly describing the vulnerabilities of Bluetooth.

The second section aims at throughly explaining the Double Ratchet algorithm, in order to make the reader familiar with it. This is strictly necessary, as said details are crucial to fully understand the following points and the related challenges.

The third and fourth sections focus on the two main contributions offered: a high level implementation of the aforementioned Double Ratchet algorithm at the Application-Layer, and the findings on possible structural attacks on Double Ratchet.

Finally, the fifth section describes a Signal-app specific attack on disappearing messages: this is not related to Double Ratchet or Bluetooth, but it is interesting nonetheless.

Chapter 2

Background

2.1 Vulnerabilities of Bluetooth

In order to properly understand the motivations behind this project, it is first necessary to know the Bluetooth technologies and their vulnerabilities. Bluetooth offers two different technologies:

1. Bluetooth Classic (BT): offers high throughput persistent connections (used by speakers, headphones, cameras etc.).
2. Bluetooth Low Energy (BLE): compared to BT, very low energy consumption (used by fitness trackers, contact tracing, etc.).

Both of these technology use a pairing key and a session establishment mechanism to try and provide secure communication. The aforementioned protection, however, is not up to date with modern standards. Devices are vulnerable to standard-compliant attacks such as:

- BIAS attack [1]: can be performed on BT and targets Bluetooth session establishment to allow an attacker to impersonate any device.
- KNOB attack [2, 3]: can be performed on both BT and BLE and allows to downgrade the security of the established keys.

The attacks break basic cryptographic properties, such as integrity and confidentiality. Indeed, this is worrying on such a widespread and pervasive technology. As a consequence, our minimum goal is to find some mean to provide these basic properties to Bluetooth-based communication. As will be shown in the following section, however, even more modern and advanced properties can be implemented to make Bluetooth secure-by-design, bringing it up to date with communication technologies that offer state of the art protocols for that aspect.

2.2 Double Ratchet algorithm

2.2.1 Overview

The Double Ratchet algorithm is a cryptographic algorithm designed in 2013, and currently used by a large number of messaging app: among those are WhatsApp, Facebook Messenger, Signal, Skype, and many more. Thanks to the elaborate combination of many cryptographic primitives, the Double Ratchet algorithm provides basic properties such as confidentiality, integrity, and authenticity. In addition, however, the following more modern and advanced properties are satisfied:

1. Forward secrecy: session keys are safe even against an attacker that learns a long-term secret used to compute them.
2. Future secrecy (also known as post-compromise security): even if the secrets of one party have already been compromised, a security guarantee about the communication still stands.

In short, the first of these two properties offers guarantees on confidentiality of *past* messages, while the second one lets the algorithm self-heal, preventing attacks on *future* parts of the communication. Indeed, these are truly desirable properties, that allow for the deep level of security that are required for a technology such as Bluetooth.
In order to fully understand the contributions detailed in the following sections of this report, it is now necessary to dive deep into the Double Ratchet algorithm, opening the black box and analysing it at a lower level of abstraction. Indeed, the provided explanation does not aim at being exhaustive, as its main purpose is to provide the reader with enough information to understand the following sections: indeed, additional details can be found in the official documentation [4]. Those already familiar with the algorithm can instead skip to Section 4.

2.2.2 Technical explanation

As the name itself suggests, the Double Ratchet algorithm is first and foremost based on two ratchets:

1. Symmetric ratchet: used for the computation of encryption key (ek), authentication key (ak) and initialization vector (iv). These are then used to encrypt/decrypt messages.
2. Diffie-Hellman ratchet: used for the initialization of the Symmetric ratchet.

Before further exploring the functioning of these ratchets, however, it is obviously necessary to define the concept of ratchet itself. A *ratchet* can be defined as a *KDF chain*. A *KDF* (Key

Derivation Function) is a cryptographic primitive that takes a secret KDF key and some data as input, and provides a single output. Assuming the key is secret and random, the output data is random-like to an observer. In a KDF chain, the output of the KDF is split into two parts: one is used as input key for the following KDF, while the other is treated as an actual output.

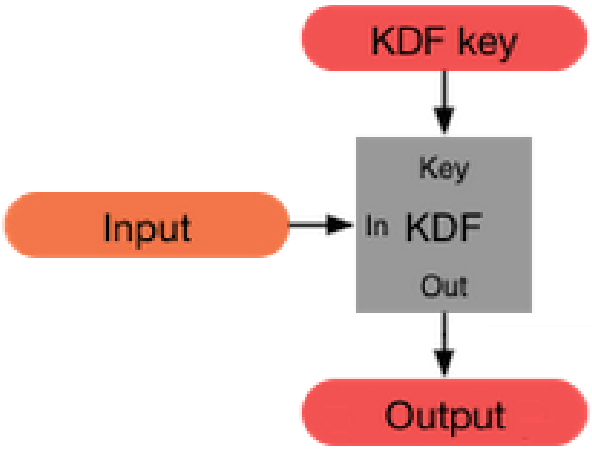

Figure 2.1: Visualization of a KDF adapted from [4]

Symmetric ratchet

The *symmetric ratchet* is a ratchet that takes as input a chain key (ck_0) and a constant, and in the ratchet step derives the next chain key (ck_1) and outputs a message key (mk). mk is then used as input for a HKDF, which outputs data that is split to obtain ek, ak, and iv. As a consequence, each ratchet step allows for the encryption/decryption of a single message. In particular, two symmetric ratchets are present for each user, one for encryption, the other for decryption.

Indeed, thanks to the combined properties of KDFs and KDF chains, the symmetric ratchet alone offers the following properties:

1. Confidentiality: ek and iv are random and secret per KDF definition, assuming ck_0 is random and secret.
2. Integrity: ak is random and secret per KDF definition, assuming ck_0 is random and secret.
3. Forward secrecy: as it is not possible to move backwards in a KDF chain, leakage of a chain key ck_n does not allow an attacker to decrypt messages encrypted with a ratchet step taking as input key $\{ck_0,...,ck_{n-1}\}$.

Since the input used along with ck is constant, however, no future secrecy property is guaranteed by the symmetric ratchet alone: in case a ck_n is leaked, all future messages can be decrypted

if the constant has been found. In order to understand how the algorithm manages to self heal, the functioning of the Diffie-Hellman ratchet, as well as its interaction with the symmetric one, must be illustrated.

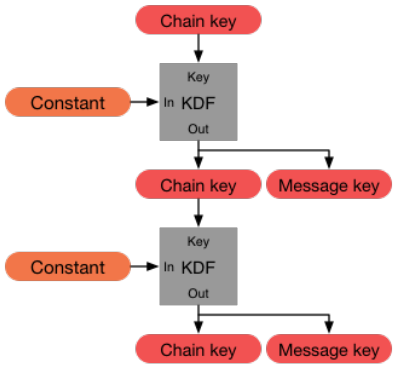

Figure 2.2: Visualization of the symmetric ratchet [4]

Diffie-Hellman ratchet

The Diffie-Hellman ratchet is based on the computation of a Diffie-Hellman shared secret between the two users. In the standard Double Ratchet algorithm, fresh Diffie-Hellman key pairs are computed each time a non-consecutive message is received, and the latest public key is attached to each message sent as part of the header.
Diffie-Hellman shared secrets are used as input for the Diffie-Hellman ratchets along with a root key rk, which represents a long-term shared secret that the two parties must agree upon before the initialization. A Diffie-Hellman ratchet step derives the following rk and output a ck, that is subsequently used in one of the two symmetric ratchets.

Understanding the interaction between Diffie-Hellman ratchet and symmetric ratchet is key to explaining the future secrecy property: as chain keys are often replaced, acquiring a chain key only allows to decrypt messages until a Diffie-Hellman ratchet step is computed.

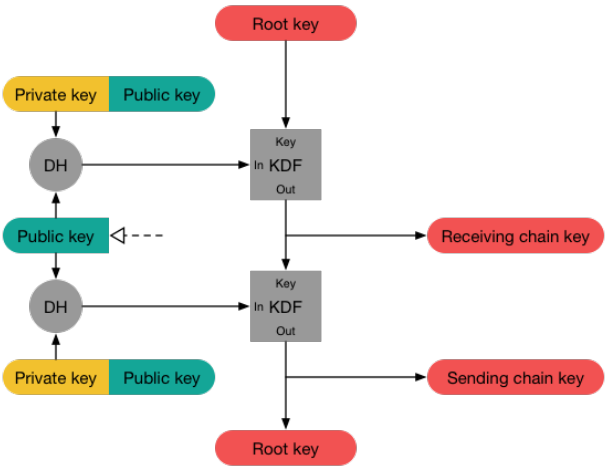

Figure 2.3: Visualization of the Diffie-Hellman ratchet [4]

Out-of-order messages

Now that the basic functioning of the Double Ratchet algorithm has been detailed, understanding of the way out-of-order messages are handled is the last requirement to fully understand one of the attacks that will be described in the following sections.

In order to make sure no message is lost even if some arrive out of order, each header contains the following three fields:

1. N: a progressive index that represents the position of the message in the current symmetric ratchet chain. The first message sent will have $N = 0$. N is reset every time the chain key is re-initialized by the Diffie-Hellman ratchet.
2. PN: an integer that represents the number of messages in the symmetric ratchet that used the previous public key. Allows to compute the necessary ratchet steps in the symmetric ratchet, before re-initializing the public key.
3. Diffie-Hellman public key: current Diffie-Hellman public key of the sender, as mentioned in the previous section.

As both N and PN parameters are necessary in order to compute ek, **the header is not encrypted, but only integrity-protected**. This is an important detail to keep in mind for the following sections. When a late message arrives, mk are computed and stored until the symmetric ratchet reaches the required index. This is indeed a necessary step, as the KDF chain cannot move backwards, and the keys would otherwise be lost, making late messages impossible to decrypt.

Chapter 3

Contribution: high level implementation of Double Ratchet on top of BluetoothChat

3.1 Design

The first goal of the project was to evaluate whether or not the Double Ratchet algorithm could be implemented to improve Bluetooth security. This allows for two possibilities:

1. Link-Layer implementation, using softwares such as Bluedroid or Fluoride to interface with the Bluetooth Stack.
2. Application-Layer implementation, assuming an untrusted Link-Layer, as it is in the current standard.

Ultimately, it has been decided to start by implementing secure data exchange at the Application-Layer. Android has been chosen as the target platform, because of the higher amount of resources available. In particular, the algorithm has been implemented on top of a sample open-source app offered by Google, known as BluetoothChat. First and foremost, the app aims at implementing the Double Ratchet algorithm as presented in the Signal specifications. When feasible, the interfaces have been kept consistent with the aforementioned specifications, in order to allow for a clear and immediate understanding of the implementation from people familiar with the algorithm. Necessary parameters that were not explicitly spelled out in the specifications were adapted from Signal's open source code, when possible.

3.2 Implementation

The application allows two users to safely communicate with messages encrypted through the Double Ratchet algorithm, thus giving positive answer concerning the possibility of making Bluetooth secure through an Application-Layer implementation. All of the suggested interfaces have been successfully implemented and work according to the defined standards.

When possible, Java's standard cryptography library was used. However, SpongyCastle (Android-compatible implementation of BouncyCastle) had to be used for Elliptic Curve Diffie Hellman related operations.

For observability's purpose, standard serialization methods have not been used for objects such as keys. Instead, byte representation of each piece of data is extracted, along with constructor overloading to create objects.

Compatibly with the standards, a Diffie-Hellman step is computed each time a non consecutive message is received. As the Diffie-Hellman step makes the encrypting operations slower the , only applying Diffie-Hellman step after an agreed number N of non consecutive messages received.

The following implementation choices have been made in order to fully focus on the crucial cryptographic aspects of Double Ratchet:

1. The initial shared root key has been hard-coded. Specifications suggest the use of X3DH (three way Diffie Hellman), but its implementation is impossible in absence of a server. An authenticated key exchange between the two parties could be designed and implemented.
2. The role of a user (i.e.: whether they are the initiator or not) is hard-coded. The role could instead be determined in other ways (e.g.: master or slave is the initiator).
3. Constraints are imposed on the length and rate of messages. A simple application-specific header that includes the number of bytes to read would solve the issue.

Chapter 4

Contribution: structural attacks on Double Ratchet

4.1 General attack setup

In order to properly test some of the attacks, two Android A40 phones have been rooted in order to allow the installation of a Frida server. Dynamic testing through Frida allowed for much more efficient troubleshooting on attacks, that would not have been otherwise possible through simple static analysis.

Rooting of the first Android A40 phone with Android 10 operating system was not trivial, as the initial procedure followed led to the phone being bricked. After unbricking it, a tutorial for Android 9 was adapted. The same procedure was then applied for the second phone.

4.2 DoS attack on out-of-order messages v1

As pointed out in Section 3.3.3, the Double Ratchet specifications suffer from a TOCTOU bug. As the information contained in the header is used before its integrity is checked, a malicious attacker might try to manipulate it with the goal of damaging users.

Let there be two users, namely Alice and Bob, who are communicating on a channel protected by Double Ratchet based encryption. In addition, let there be a third user, Charlie, who acts as a man in the middle and can modify data passing through him. Finally, Charlie aims at crashing the app for one of the two users (namely, Bob in this case). In order to do this, Charlie can use the fact that the Double Ratchet algorithm stores one 32-byte mk for each skipped message, and try to fill the device's memory. This would happen before the ak can be computed, and, therefore,

before the integrity check can happen.

4.2.1 Design

In this first variation, the attack is fairly simple: let Alice send a message to Bob such that $N = 1$. After intercepting the message, Charlie edits the data such that $N = 10^9 + 1$. As a consequence, Bob will have to store 10^9 mk in order to reach mk_{10^9+1}. As each mk is 32-bytes long, this would require almost $32GB$ of memory, almost certainly crashing the device.

4.2.2 Implementation

The first version of this attack has only been thought of, but never actually implemented. Static analysis of the open-source code of the Signal app on Android suggested that only up to 2000 out-of-order key derivation steps are computed for one out of order message. As a consequence, a more clever strategy had to be conceived.

4.3 DoS attack on out-of-order messages v2

4.3.1 Design

The attack uses the same premises and setup as the previous one. In this case, however, Charlie increments N by m, with $1 < m < 2001$. As the message arrives, Bob will compute m symmetric ratchet steps, storing $m-1$ keys in the meantime. The integrity check will then take place and fail. If such failure does not lead to deletion of out-of-order keys and/or resetting Bob's symmetric ratchet index, Charlie can repeat the process, eventually leading to a failure in Bob's device.

4.3.2 Implementation

Implementation has first implemented in the testing framework of Android Studio, on the Bluetooth Chat application. Despite working as intended, the framework memory management system somehow allowed the program not to crash. As a consequence, the same attack was implemented by editing the sample app itself, so that a single message from a user could be automatically sent multiple times with an increasing value of the header. This worked as intended, crashing the victim's app. Finally, the attack was tested on the Android version of Signal using a Frida script, which edits the message header as it is about to get sent. The test, however, yielded a negative result, as the app's session management system deletes out-of-order keys if a message does not pass the integrity check.

4.4 DoS attack on out-of-order messages v3

In order to implement the third variation of this attack, an alternative set-up is needed: while the same channel between Alice and Bob exists in this case, no man in the middle is present. On the contrary, Alice is a malicious user, and wants to crash Bob's application.

4.4.1 Design

In terms of header manipulation, this attacks is no different from the previously presented one. In this case, however, Alice unilaterally advances her symmetric ratchet by m steps: as a consequence, the message is legit and will pass the integrity test, but will still be considered out of order.

4.4.2 Implementation

The attack has only been tested on the Android version of Signal using Frida, as it only slightly differs for the previous one, and aims at circumventing a mitigation. In this case, Alice's key derivation function has been hijacked and looped m times through Frida, Once again, however, the result was negative: dynamic testing shows that Signal actively put a limit to the number of stored out of order messages, thus completely mitigating this attack.

4.5 Impersonation attack

4.5.1 Design

The threat model for this more sophisticated attack requires once again two users, namely Alice and Bob, to be communicating through a channel implementing the Double Ratchet algorithm. In this case, the attacker, Charlie, managed to clone the device of one of the two users (namely Alice), and can therefore communicate with Bob posing as Alice. The goal is to be able to perform a long-term impersonation of the victim, while also cutting her from the conversation.

If Charlie sends a message to Bob before Alice does, Bob's and Charlie's symmetric ratchets (receiving and sending, respectively) will move at the same time, while Alice's will be left behind. As a consequence, Bob will not be able to read Alice's messages, but the converse is not true: as Bob's sending symmetric ratchet and Alice's and Charlie's receiving ones did not execute any step, Bob's message will reach both targets. As a message from Bob is received, both Charlie and Alice will need to compute their own Diffie-Hellman key pair in order to reply. In this case, the

Diffie-Hellman public key of the first to reply is used to compute the Diffie-Hellman ratchet step, effectively cutting out of the conversation the slowest between Alice and Charlie.

4.5.2 Implementation

The attack has been implemented in the testing framework of Android Studio, on top of the BluetoothChat sample app. With the defined timing, the attacker can fully complete the impersonation attack.

4.6 Disappearing messages attack on Signal app

A final attack has been thought of and designed while performing dynamic analysis on the Signal app for Android. This attack does not have any relation with the Double Ratchet algorithm, and is rather an extra attack that was found along the way.

4.6.1 Design

Signal's disappearing messages feature allows a user to automatically delete messages in the chat for both user after a set amount of time. For the threat model, let there be two users, namely Alice and Bob. Let us suppose Alice set a timer t, after which messages will disappear in both users chat. Let Bob be a malicious user: his goal is to keep all the messages for an indefinite amount of time, while Alice does not notice this change, as messages in her chat disappear.

4.6.2 Implementation

The attack has been directly implemented on the Android version of Signal using Frida. By intercepting the message that Alice's device automatically sends when setting the timer, the script reduces it to 0, effectively performing the attack.

	Sample app	Signal
DoS attack v1	-	-
DoS attackv2	✓	X
DoS attack v3	-	X
Impersonation attack	✓	-
Disappearing messages attack	-	✓

Table 4.1: Summary of the attacks' results

Chapter 5

Conclusion

Given the work done throughout the project, the following conclusions can be drawn:

- The Double Ratchet algorithm can be implemented at the application layer to allow secure data exchange on top of Bluetooth, as seen with the BluetoothChat application. Considering the strength of the protocol, it is worth working on a Link-Layer implementation.
- Despite its structural strength, the Double Ratchet algorithm is subject to several attacks, due to elements that are considered implementation details. This includes both simple DoS attacks and more elaborate attacks, such as the impersonation presented above.
- The Android implementation of Signal is resilient to the identified DoS attack. However, it is worth pursuing all of the described structural attacks on many apps, both open sourced and not.

Bibliography

[1] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. “BIAS: Bluetooth Impersonation AttackS”. In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. May 2020.

[2] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. “Key Negotiation Downgrade Attacks on Bluetooth and Bluetooth Low Energy”. In: *ACM Transactions on Privacy and Security (TOPS)* 23.3 (2020), pp. 1–28. DOI: 10.1145/3394497.

[3] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. “The KNOB is Broken: Exploiting Low Entropy in the Encryption Key Negotiation of Bluetooth BR/EDR.” In: *Proceedings of the USENIX Security Symposium*. Aug. 2019.

[4] *The Double Ratchet Algorithm*. https://web.archive.org/web/20210115164528/https://signal.org/docs/specifications/doubleratchet.