Evaluation of the security of smartwatch communication

Harm Dermois, James Gratchoff, Florian Ecard, Master SNE, UvA

December 2014

UNIVERSITY OF AMSTERDAM

Abstract

Contents

1 Introduction

With the uprising of IoT (Internet of Things), more and more wearable devices start to get connected. One of these wearable devices is the smartwatch. It is mostly used to display informations or notifications coming from a phone through Bluetooth communication. It can also be used to collect data. The data exchanged through the air is most of the time private information and could be sensitive. That is why it is important that this data is secure.

With all new technology comes new security concerns. Security issues are even more relevant in the case of small battery reliant devices. As encryption and security can be very computation intensive, this will take huge toll on battery lives of these devices. The aim of this project will be to investigate how secure the communication between a smartwatch and a smartphone is, and try to find solutions or alternatives to the problems found (if any) in our investigation.

This project has been conducted with an Android phone (version 4.0 or plus) that will act as the master and a Sony SW2 Smartwatch that will act as a slave. This smartwatch is one of the newest and most available on the market. To gather information, a Bluetooth sniffer named Ubertooth One has been used.

This paper will firstly present in the literature review how the Bluetooth technology works and also introduce the tools used for this research. The methodology followed during this project will then be presented. An analysis of the results will outline the possibility of hacking a Bluetooth device and why we couldn't hack it ourselves. Finally, the conclusion and possible future work will be described.

1.1 Research Question

Information exchanged between smartwatches and smartphones is private and needs good security in order not to leak any sensitive information. The main means of communication between smartwatches and smartphones is Bluetooth. The use of low-power communication makes it more likely that there will be security issues with the communication and that encryption might not be set up as it is asking a lot of computation resources. This paper will try to answer these questions:

> How secure is the Bluetooth communication between smartphone and smartwatches?
>
> Is it possible to eavesdrop any data exchanged during this communication?
>
> Is it possible to change the content of the data exchanged?
>
> Is it possible to take control of a smartwatch and/or the smartphone?
>
> How the vulnerabilities found (if any) could be addressed?

1.2 Ethics

From an ethical point of view, there will be no major issues for the experimentation that will be conducted. The equipment and the data used will be the authors/university property. The authors will not eavesdrop other people devices that are not involved in the experimentation. The publication of our paper could bring ethical issues as the smartwatch market is expanding and indeed, if this research comes up with security flaws they could be used in an evil way by others. That is why the paper produced will evaluate the ethical problems brought by the research (if any).

2 Literature Review

The inspiration for doing this project came from *this paper* [?], it explains how BTLE works and how it can be exploited. In the Future work part, Ryan talks about doing a man in the middle attack, but in a recent talk he had [3], it has been said that this was already done.
However, this kind of Man-in-The-Middle (MiTM) attacks wasn't performed on the 3.0 version, as this version is known to be more complicated, this is why we chose to look into it by trying to break into a smartwatch.
Another paper we used to be sure of our choice is *this* [2], which gives an explanation about Bluetooth, how it works, its security mechanisms and tools to use to exploit and inspect Bluetooth.

2.1 Bluetooth

Bluetooth is a wireless communication technology created in 1994 by the mobile telecommunication company Ericsson. Bluetooth was designed for low-power consumption devices such as sensors, mobile phones, etc. Nowadays, more and more devices use Bluetooth mainly due to the uprise of Internet of Things (IoT) and small battery reliant devices.

2.1.1 Bluetooth communication

Bluetooth operates in the 2.4 GHz frequency band and uses frequency-hopping spread spectrum. This FHSS means that each packets from a Bluetooth communication is transmitted on one of 79 channels having a bandwidth of 1MHz each.
Bluetooth is also using adaptive frequency-hopping spectrum (AFH), used to avoid crowded frequency in the channel spectrum.

In order to fully understand Bluetooth technology, older versions need to be described. The difference between versions were based on the *"versions differences" from links* [XXX] and *Wikipedia*[XXX].
In versions 1.X (released between 1994 and 2005), the transmissions could go to a speed up to a theoretical value of 1Mbps. Flow control and retransmission modes for the Logical link control and adaptation protocol (L2CAP) were added in the latest version (v1.2). These versions are now obsolete and nearly extinct.
Version 2.0 + EDR (Enhanced Data Rate) was released in 2004, it is the first version implementing this technology, speeding up the maximal data rate transfer to 3Mbps rather than 1Mbps.
The version 2.1 + EDR (2007) is not bringing any new speed but despite that, its major improvement is to considerably improve the security of Bluetooth. Indeed, SSP (Secure Simple Pairing) is introduced in order to make the pairing process simpler, quicker and safer. More explanations about SSP are given in the next part.

The case study in this project is however not using this latest version, but the Bluetooth 3.0+HS (High Speed) which was released in 2009, this version is the first one providing a high data transfer speed that can go up to 24 Mbit/s. It is able to provide such a speed by using the famous 802.11 wireless protocol, it uses a ”go all out” policy which first uses normal Bluetooth communication for pairing process and when a file is wanted to be exchanged or whatsoever, it swaps to the wireless in order to provide this speed.
Currently, the latest version of Bluetooth is 4.1 which is known as Bluetooth Low Energy (BT-LE). As its name says, BT-LE was designed to reduce power consumption while providing the same communication range and speed that were used in the previous versions. This version also is simpler than the 3.0 and uses AES to encrypt data. However, it is less secure due to its simplicity and this version has proven MiTM attacks [XXX].
The information provided further in this report about Bluetooth technology is concerning the version 3.0+HS, although it may be applicable in other versions as well.

2.1.2 Device ID

The Bluetooth address (BD_ADDR) of a device is composed of 48 bits.
This address is divided into three parts, the first one being called the UAP (Upper Address Part) which is 8 bits long and, combined with the second part, the Non-significant Address Part (NAP, 16 bits long), they form together the manufacturer ship, or company_id. The last part is 24 bits long and is the Lower Address Part (LAP), almost uniquely identifying a device, this one will be used in the first stage of sniffing in order to decode the captured packets (This will be further explained later on). This is shown in the figure 1 below.

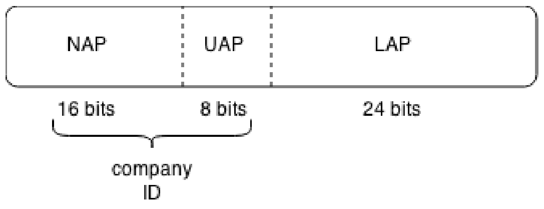


Figure 1: Bluetooth address definition

2.1.3 Overview of Bluetooth security

The basic security in Bluetooth is set up by the user, he has the choice between three different modes:

- Silent: In this mode, the device will not initiate any connection. The only thing it does is monitoring the traffic.
- Public: The device is discoverable by anyone around it having its bluetooth activated.
- Private: The device will in that case only answer to device that it already paired with, thus making it (theoretically) only discoverable to known devices.

A Bluetooth device can implement four different security modes:

- Non Secure: As its name suggests there is no security.
- Service-level enforced security mode: It establishes a non secure ACL (Asynchronous connectionless) link between the two devices willing to communicate. In order to introduce optional encryption, authentication and authorization, a request has to be made via L2CAP (Logical Link Control and Adaptation Protocol) connection-oriented or connection-less channel. This mode is used in versions 2.0 + EDR and below.
- Link level enforced security mode: Once the ACL links are done then security procedures are initiated.
- Service-level enforced security mode: This one is very similar to the second mode except that it introduces SSP (Secure Simple Pairing) and this is compatible only with Bluetooth versions from 2.1 + EDR.

2.1.4 Pairing process

As briefly explained earlier, SSP has been implemented in order to overcome the weakness of the 4 digits pass at pairing time (only about 10.000 possibilities max) in addition to making the pairing process easier. Indeed, using an elaborated Bluetooth sniffer would allow to almost instantly find these digits and then bypassing the security.

Two devices pair by creating a shared key, called the Link key. This key will be used to authenticate and encrypt the data of two devices. In order to do so, there are two possibilities: LMP-pairing (4-digits/pincode manner) or SSP.
Once this done, the two devices can store their mutual key in order to use it later for a re-pairing, making the connection much faster.
In LMP-pairing, the only thing that is not transmitted through the air is the 4 digits PIN code. The pairing process in this case works as follows:
The initiator will first generate a 16-bytes random number and then send it to the other device. Once received, both users will put in their PIN code which they will use to create an initialization key, the later will be converted into the actual LMP key. Finally, the two devices have to authenticate each other with their respective newly created LMP key.

A software created by Ellisis [XXX] allows an attacker that listens to the traffic to guess with the given information the 4 digits, thus making LMP useless.
An overview of the previously explained LMP pairing process is shown figure 2:

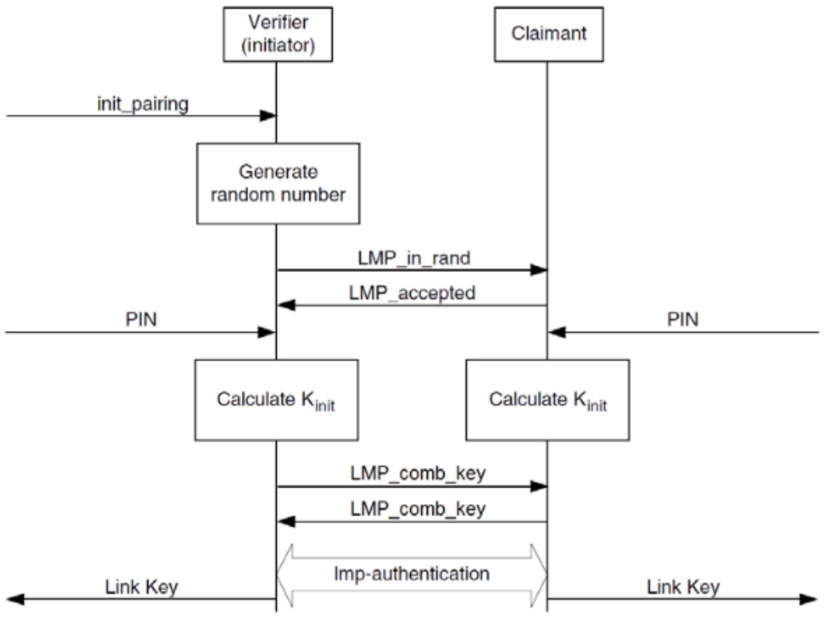


Figure 2: LMP pairing process

`http://www.ellisys.com/technology/een_bt07.pdf`

SSP was then created, fixing the above vulnerability even though it is still possible to find this mechanism in many devices for backward compatibilities. SSP is a new manner of generating a Link key by using Elliptic Curve. With this, it is possible to create a much bigger random number allowing a range of possible Link keys is 2^{128}. This number is far too big to be brute forced.

In order to create a public/private key pair (DHKey), SSP uses the famous Diffie-Hellmann algorithm with a 192-bits random number. The public key is transmitted over the air and can be seen by anyone, the private key is however kept secret for obvious security needs. Using two given key pairs A and B, there is a function that allows to result in a DHKey and only the two devices should be able to find it out. This function is F(Public A, Private B)=F(Public B, Private A). This very DHKey will be used in order to create the link key in between the two devices.
From there, the rest of the pairing process is similar to the LMP-pairing.

SAFER+ and protection against MiTM with SSP

While BTLE uses AES, the older Bluetooth versions up to the 3.0+HS are using SAFER+ (Secure and Fast Encryption Routine) in order to provide authentication and key generation.
SAFER+ was candidate to the AES process and like it, it is a family of block ciphers using rounds and a block size of 128 bits.
In Bluetooth, SAFER+ is used for key derivation and authentication.
This algorithm uses a non-orthodox linear transformation in order to provide diffusion. It also use additive constant factors in order to avoid weak key generation.
Like in AES, it has two main components. The key-scheduling unit which is used in order to provide the round keys on the fly. The second one is called encryption data path, the latter will mix the plaintext with the "feedback" data from the previous rounds.

In order to be protected against MiTM attacks, the device can use the user (e.g. to compare two numbers) or Out Of Band (OOB) channel (NFC for example, used to check the integrity of a checksum), these two cases are two of the four association models of SSP. Let's define these models:
The numeric comparison association model which is the first case above, asking the user to compare numbers.
The Passkey Entry association model which is used when one of the device has input capabilities but no displays, in this case only the user with input capabilities will answer the 6-digit pass code.
Finally, if one of the device has neither input nor output, and that OOB can't be used, then the JustWork association model is used, which is the weakest model as it simply asks the user to accept the connection.
In SSP there are 6 different phases in the pairing process:

1. Capabilities exchange: discovering devices or re-pairing ones will exchange their input/output capabilities.
2. Public key exchange: Compute the private/public key using Diffie-Hellmann and give to each other their public keys.
3. Authentication stage 1: The protocol will depend on which of the four association model has been chosen. The aim of this stage is to be sure that nobody can eavesdrop the connection.
4. Authentication stage 2: There, the devices have exchanged their data and the integrity of each other is verified.
5. Link key calculation: Using the previous steps, the Link key can be calculated (the DH key and nonces). It also needs the BD_ADDR.
6. Authentication and encryption: Now the encryption keys can be generated in order to encrypt all the data that will be exchanged between the two devices.

The information provided in this section are based on K. Haataja et al. book [XXX] and wikipedia [XXX].

2.2 Ubertooth

Bluetooth technology is harder to sniff compared to other wireless protocols such as 802.11 that already have solutions for promiscuous packet sniffing. Because of how the way Bluetooth works, it is harder to tap the communication between two devices. The entities need to pair before being able to send real messages. Normally, it is only possible to see the discovery messages which are used to find out which Bluetooth devices want to be found. Another problem is that Bluetooth devices can choose to stay hidden.
Due to frequency hopping it is hard to keep track of a device and listen to packet exchanged. However with the Ubertooth project it is now possible. To do so, 3 parameters need to be known prior to be able to retrieve any information: the Lower Address Part (LAP), the Upper Address Part (UAP) and the clock (CLKN) that is the upper 26 bits of the CLK27 of the masters clock. The process to find this parameters is explained later on.
Mike Ossman started his project in 2010 under the name of Ubertooth. This device is dedicated to create and build an open source hardware that is able to follow and sniff Bluetooth communication. The Ubertooth was initially made to do following of Bluetooth devices communications which can also be seen in the way the plug-ins for Kismet work and the names of the tools made for the Ubertooth. The tools will be explained in section 3.1. The latest release (Ubertooth 1) is capable of sniffing communication by identifying the right address and clock. The following sections explain how the clock and address are found and implications of finding these.

2.2.1 Finding the UAP

The UAP is the most important in order to do anything interesting with Bluetooth. For instance it can determine the hopping sequence that is used for the communication between two devices, it represents 8 bits of the 48 bits BD_ADDR. The LAP is transmitted in plain text, this makes it easy to find as it is present in every packets. The First part of the address is the NAP which is ignored because it is not needed for the initial communication. So only the UAP is needed and as it is only 8 bits, it can be brute forced. However, this is not a good way to find it as brute forcing is detectable and it will only work when the master is in a connectible state. The following is discussing several methods that Ubertooth can use to find it out.

The Ubertooth is made to be a passive sniffer so the brute force method is not used. Instead, one of the techniques used to determine the UAP is by using the HEC(Header Error Check), that will check if the header is received without errors. With this it is possible to determine what the unknown bits are, which will be the UAP. This header is sent with each packet so it makes it easy to decode but the problem is that it is XOR with a pseudo-random stream, this is called "whitening". To decode this whitening, there are 64 possible streams that will be tried depending on which master clock is used.

The lower 6 bits of a clock are used to determine the master clock and also to make the hopping pattern. It will then generate 64 candidate UAP with each of the 64 streams. Sometimes the packets will have a CRC (Cyclic Redundancy Check) that can be used to check if the right stream is used to decode that packet.

Another method is to perform a series of sanity checks on the packet format, allowing the Bluetooth address to be unwhitened. If the packet type is known, some information can be derived such as the packet length. These information can confirm or deny a possible match. The problem with this is that false negatives can happen and it can eliminate UAP streams which are valid. This can happen, because the data that is decoded will be of wrong packet type and it will make the wrong conclusion on whether to keep or to throw away this possible clock. This will lead to not finding any UAP for the master and the process will have to be restarted in order to find it.

2.3 Sony Smartwatch SW2

The smartwatch that was chosen to do the project with is the Sony SmartWatch 2. The SW2 uses Bluetooth 3.0, it is relatively cheap and programmable. The SDK has recently been updated and is easy to use. The SDK was used to make to make apps for the SW2. The installation guide can be found at this website [1]. The SDK contains examples. These examples were used to generate traffic for the Ubertooth to sniff. The Ubertooth needs a decent amount of packets decrypt the packets. Another benefit of using a self made app is that you exactly what is the content of the packets. This makes it easier to understand the traffic.

Figure 3: Sony Smartwatch 2

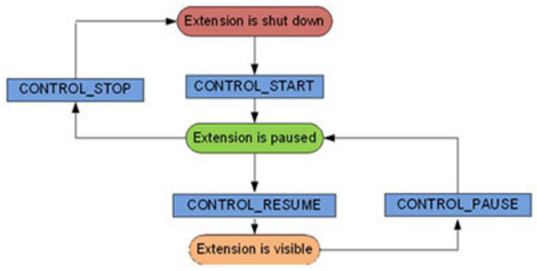


Figure 4: Life cycle of a Sony smartwatch app

https://developer.sony.com/develop/wearables/smartwatch-2-apis/guides/build-your-app/

All the Sony devices use an application called Smart Connect. This application keeps track of all the devices that want to make contact with the phone. It also keeps track of the applications that are installed on the watch. Whenever the pairing is stopped between the phone and the watch, all the previously installed applications are removed from the watch until these two devices are paired again.

2.3.1 Smartwatch applications

The life cycle of a smartwatch app is about the same as an application on a phone or tablet. This can be seen in figure 4. As stated before, the smartwatch apps only work when they are paired with the master device(the device which having the Smart Connect installed). When making an app, a few things are needed. First, register the app because it is needed for Smart Connect to find the app. **Register it to whom/where?**

Then update the AndroidManifest.xml file that defines the programmer main activity and some control features. After this, the app can be coded by including the mandatory classes to make the application compatible with SW2.

Sony has made a level of abstraction that makes it easy to send messages using intents. As stated before, the pairing process is handled by the Smart Connect.

3 Methodology

This section describes the methodology followed to perform our analysis on Bluetooth communication. First the tools that the Ubertooth has will be discussed. The next to parts will present how we investigated the different layers to do the analysis more efficiently. The last part will present the methodology followed to investigate the pairing process between a phone and the SW2.

3.1 The Ubertooth Tools

The Ubertooth comes with some tools to make its use easier, these tools are included in the host code which is provided by the project Ubertooth. The main Ubertooth tool we are using is `ubertooth-rx`. `ubertooth-rx` listens to all the packets sent on the bandwidth and displays them. The default functionality is: Scan the Bluetooth bandwidth and try to look for all the UAP of the devices. By using the option `-l <LAP>` the tool will do a more in depth analysis. It will then use the given LAP and follow it to try to find the UAP and the clock which will be explained in 2.2.1. When the program has found these two values, it will have enough information to decrypt the packet. This packet will then be printed with all the information, being exactly where what is thus research about.

There is also a Kismet plug-in available which also uses `ubertooth-rx`, but displays it in a better way in the Kismet interface. It also makes log files in pcap format which can then be used to analyse the traces in Wireshark. The Kismet plug-in can only be used to discover all the Bluetooth devices in your neighbourhood and find their UAP. After it found the UAP it will continue by trying to find the clock.

Tool	Description
`ubertooth-rx`	Passively scan for all the Bluetooth devices and try to find the UAP for them.
`ubertooth-scan`	An active scan of all the Bluetooth devices in the neighbourhood.
`ubertooth-btle`	The Ubertooth low energy tool.

Table 1: Ubertooth commands used during this project

3.2 The Bluetooth stack

The Bluetooth stack has many layers. For this project the important layers are the HCI, LMP, and the physical layer(Bluetooth Radio). As seen in figure 5. The LMP have already been discussed in an earlier section. The HCI and the physical layer will be discussed below.

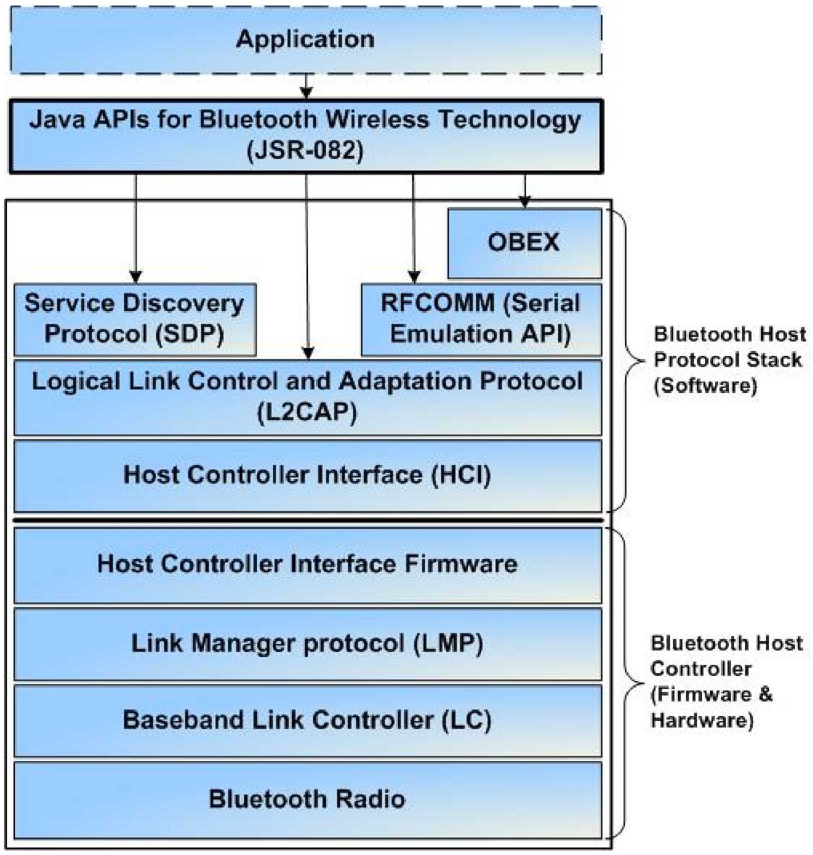

Application
Java APIs for Bluetooth Wireless Technology (JSR-082)
OBEX
Service Discovery Protocol (SDP)
RFCOMM (Serial Emulation API)
Logical Link Control and Adaptation Protocol (L2CAP)
Host Controller Interface (HCI)
Bluetooth Host Protocol Stack (Software)
Host Controller Interface Firmware
Link Manager protocol (LMP)
Baseband Link Controller (LC)
Bluetooth Radio
Bluetooth Host Controller (Firmware & Hardware)

Figure 5: The Bluetooth stack

http://www.oracle.com/technetwork/testcontent/fig1-1-158000.jpg

3.2.1 Physical layer

In order to eavesdrop a Bluetooth communication, the Ubertooth One presented above has been used. This tool is able to decode packet from a targeted Bluetooth communication.

The LAP was known for the two devices. As explained in the literature review, the Ubertooth tool retrieve the important parameters (UAP, clocks) used to sniff a connection thanks to that address. Once the tool was able to sniff and follow the communication between the two devices it was possible for it to decode packets. In order to have relevant information the authors conducted experiments in a certain way. Indeed several captures have been performed recording:

- The pairing process between the two devices
- The de-authentication process between two devices
- The exchange of different data format (videos, pictures, notification)

3.2.2 HCI layer

Due to the increasing use of Bluetooth; Android, since Android 4.4, has added a functionality for developers to sniff Bluetooth packets at the HCI layer. This tool allows to monitor Bluetooth traffic in forms of pcaps logs for Wireshark. These logs can be found in the root of your data directory and is called btsnoop_hcilog.

The HCI layer is just above the LMP layer where authentication and encryption/encapsulation takes place. This can be seen in figure 5. The data and messages exchanged at this layer are in clear text and packets are not encapsulated. This makes them easily identifiable and easy to monitor.

The experiments performed above at the Physical layer has been performed in the same way (and at the same time) at the HCI layer. Both tools need to be run at the same time to compare the data.

3.3 Comparing data

The purpose of looking at different layers is to correlate the packets from the RF with the packets from the HCI level. This is about the packets we do not know what they mean. These are the rogue packets. The POLL and NULL packages cannot be seen because they are handled in a lower level the LC as seen in figure 5. But sometimes we get packets with different types. These packets contain data and these are expected to be received in one way or another on the HCI level. To compare these rogue packets, the Ubertooth and bt_snoop from the android device have been used. The information retrieved from both of the sources is then compared on the timestamps. Unfortunately, the packets on the HCI level have a higher level of abstraction and do not correlate to the packets that are sent over the bandwidth. The assumption made is that the timestamps that is at which the packet arrives at the Ubertooth is not much different from the timestamp that is seen in HCI. Another assumption is that these packets will leave some trace in the HCI in the form of a request, data transfer or an anomaly at the HCI.

3.4 Comparing data experiment

This is the experiment that has been done to correlate the data from the Ubertooth with the data we see in the HCI:

- Install a slightly modified version of the example app called HelloActiveLowPowerActivity(the modification sends a message from the watch to android device once every second)
- Start `ubertooth-rx -l <LAP -u <UAP`. We give also the UAP, because Ubertooth will not have to find the UAP by itself.
- Then start the smartwatch app that sends the messages
- Both HCI layer and the Physical layer will be logged.

We choose an interval of one second per message, because we want to send many messages. With this many messages we have a higher chance of finding the rogue packets we are interested in. Sending it faster than this will make it very hard to correlate anything, because there will be so events at the same timestamp that it will be impossible to correlate anything.

3.5 Investigating the pairing process

During the pairing process some critical information are exchanged. Indeed as explained in the literature review, master and slave are encrypting data thanks to several parameters common to each other: a key, a salt and a clock. These parameters are exchanged between master and slave during the pairing and this is what an attacker wants to know to be able to decrypt packets and understand what is exchanged. Many papers have discussed the way to gather these information. Indeed, in order to retrieve them, an attacker would have first to see the pairing mechanism. That is why it is require to de-authenticate the user by making it believe his device is not working. Naturally the user will then re-pair himself with his device. This is where the attacker is able to see the different parameters exchanged. The pairing process is done in the Smart Connect application. This happens before any interaction with the smartwatch apps. This means that the apps have no control over the pairing process, but it is interesting to see what is sent over. To do so the pairing process has been looked at from the HCI level of the device. At the HCI it is possible to see all the communication between the smartwatch and the device. During this process the LMP parameters are sent over line between the two devices.

4 Results and discussion

In this section, the results and the problems that have been encountered during the project are discussed.

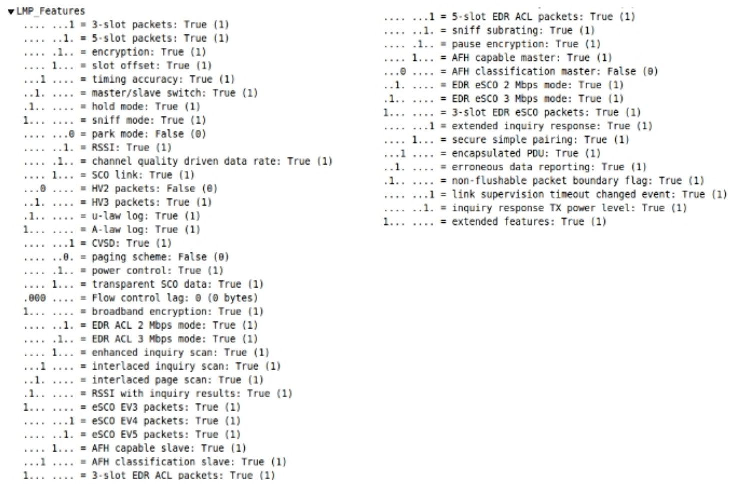
4.1 Traffic investigation

4.1.1 RF layer

At the RF layer a few messages can be seen. As shown in the appendix B the most packets seen are the NULL and POLL messages. These messages are keep alive messages. During the captures the Ubertooth is only looking at channel 0 on which these packets are sent. Some other packets are decoded, they are a few of them. These packets contains data and are of type DV/3-DH1, AUX1, AFH, DM1, HV1, DH5/3-DH5. An explanation of these packet types are explained in appendix[?]. The data found have been analysed. The only valuable information that could be output from this data analysis is that the packet are fragmented and seems to be encrypted. Indeed the packets are fragmented as they contain different LLID parameters (which specifies if the payload is the start or the continuation of a L2CAP or LMP message).

4.1.2 HCI layer

From the HCI captures, some conclusions can be drawn. Indeed, during the process, some parameters are negotiated regarding the future communication between the two devices. More specifically, what we are looking for are the LMP parameters **??**.



```
▼LMP_Features
    .... ...1 = 3-slot packets: True (1)
    .... ..1. = 5-slot packets: True (1)
    .... .1.. = encryption: True (1)
    .... 1... = slot offset: True (1)
    ...1 .... = timing accuracy: True (1)
    ..1. .... = master/slave switch: True (1)
    .1.. .... = hold mode: True (1)
    1... .... = sniff mode: True (1)
    .... ...0 = park mode: False (0)
    .... ..1. = RSSI: True (1)
    .... .1.. = channel quality driven data rate: True (1)
    .... 1... = SCO link: True (1)
    ...0 .... = HV2 packets: False (0)
    ..1. .... = HV3 packets: True (1)
    .1.. .... = u-law log: True (1)
    1... .... = A-law log: True (1)
    .... ...1 = CVSD: True (1)
    .... ..0. = paging scheme: False (0)
    .... .1.. = power control: True (1)
    .... 1... = transparent SCO data: True (1)
    .000 .... = Flow control lag: 0 (0 bytes)
    1... .... = broadband encryption: True (1)
    .... ..1. = EDR ACL 2 Mbps mode: True (1)
    .... .1.. = EDR ACL 3 Mbps mode: True (1)
    .... 1... = enhanced inquiry scan: True (1)
    ...1 .... = interlaced inquiry scan: True (1)
    ..1. .... = interlaced page scan: True (1)
    .1.. .... = RSSI with inquiry results: True (1)
    1... .... = eSCO EV3 packets: True (1)
    .... ...1 = eSCO EV4 packets: True (1)
    .... ..1. = eSCO EV5 packets: True (1)
    .... 1... = AFH capable slave: True (1)
    ...1 .... = AFH classification slave: True (1)
    1... .... = 3-slot EDR ACL packets: True (1)
    .... ...1 = 5-slot EDR ACL packets: True (1)
    .... ..1. = sniff subrating: True (1)
    .... .1.. = pause encryption: True (1)
    .... 1... = AFH capable master: True (1)
    ...0 .... = AFH classification master: False (0)
    ..1. .... = EDR eSCO 2 Mbps mode: True (1)
    .1.. .... = EDR eSCO 3 Mbps mode: True (1)
    1... .... = 3-slot EDR eSCO packets: True (1)
    .... ...1 = extended inquiry response: True (1)
    .... 1... = secure simple pairing: True (1)
    ...1 .... = encapsulated PDU: True (1)
    ..1. .... = erroneous data reporting: True (1)
    .1.. .... = non-flushable packet boundary flag: True (1)
    .... ...1 = link supervision timeout changed event: True (1)
    .... ..1. = inquiry response TX power level: True (1)
    1... .... = extended features: True (1)
```

Figure 6: LMP Parameters found during the pairing process at the HCI layer

From there, it is true to say that the communication is encrypted, encapsulated and uses SSP. It is even possible to retrieve the link keys used for encryption at this level. The link keys are always updated after a new pairing.

4.1.3 Correlation of the two layers

PULL and NULL packet are not available at the HCI layer because they are destined at the link controller layer.

The 6 other types of packets found can or cannot be correlated?

From the pairing process on the HCI, in the connection set up these packet types are disallowed. This is the reason why these packets are not found in the HCI of the master device. As you can see in appendix A. The connection settings are set only to accept these kinds of packets shown. All the packets that are send over this channel which are not poll or null are disallowed types.

The reason why these packets are sent is unknown.

4.2 The rogue packets

As already mentioned, there are some packets we cannot place. These packets can be found by the Ubertooth, but cannot be detected by the HCI. One possibility is that these packets are not meant for the master device. Since the packet types are disallowed by the master it is quite possible, but then the question is why are these sent. Another explanation is that these packets are just the same polling and null packets, but they are just badly decrypted. There is no correlation to be found by comparing the different rogue packets. In fact, there may be some correlation, but it is not clear to guess that by just looking at them as a single set. A possible solution is to have a similar tool as the Ubertooth on the smartwatch to see what is sent at the same time.
As seen in appendix C, there is an explanation of what information we get about the packet. Looking at the first packet in appendix C there are a few things that can be seen. The data is always doubled and the packet is divided in a few fields. Type, LT_ADDR, flow, payload length and data where the first line contains some information about the packet. The most important values from this information are:

- The **ch**, this is the channel in which the packet was found.
- The **LAP**, which is the sender's lower address part.

4.3 Problems encountered

There are a few problems that have been encountered during the project.
The main one is that we cannot see the most important data that is send. Indeed, the ubertooth tool that has been used, even though already powerful, cannot sniff packets that are going at high speed using the bluetooth 3.0+HS. If it had been wanted to do so, a dongle much more expensive would have been needed such as the FTS4BT costing around 5000 euros.
The data not being seeable is a very important one. Indeed, it concerns the pairing process and the data that is exchanged between the smartwatch and the android device. Using the ubertooth, the only traffic visible (after catching the UAP and the clock) are the POLL, NULL and some rogue packets that are send over channel 0.

Another problem came up when trying to correlate the data between the packets seen from the Ubertooth and from the HCI layer. The packets shown with the Ubertooth are encrypted and many packets from the HCI layer aren't displayed in the Ubertooth side. As we couldn't see the data exchanged during the pairing process due to insufficient tool capabilities, the decryption of packets were infeasible. These reasons made it really hard to find any correlation between these two packets provenance. It is expected that these packets (from Ubertooth) contain some kind of information for the HCI, but this is not certain.

The Ubertooth is a fairly new open source device and it's primary use is not what has been tried in this project. It is under active development and the tools keep improving and changing. An upgrade on the hardware of this device would be needed in order to achieve what has been tried.
In addition, only a few people are working with it, which makes it harder than usual to find a information and examples about how to use the Ubertooth.

5 Conclusion and Future work

The tools used in this project are not useful for inspecting the communication between the SW2 and another device. The only thing that can be seen are the discovery, keep alive and "rogue" packets. This makes the Ubertooth not very useful to do any kind of analysis about the traffic between the SW2 and another device. This is because the SW2 uses Bluetooth 3.0. With Bluetooth 3.0 most of the packets are send with a data rate that Ubertooth cannot follow.

On the other hand the Ubertooth can be used to the follow discoverable Bluetooth devices on the network and can uniquely identify them. The SW2 is easy is program. Ubertooth is still under development. The tools are still being improved and there is not enough documentation about how to properly use them. With time this will get better and with the if Bluetooth 4.0 stays the way it is it will be tool that can be nice to use do all kinds of fun stuff. But for Bluetooth 3.0 it is not a very useful tool and if we would want to inspect this communication there are some tools that can do this, but these are far too expensive for such a small project.

In the future experiments can be done with a smartwatch that Bluetooth 4.0 which transfer data in low energy mode. The current focus of the Ubertooth project is on the Low Energy mode of Bluetooth 4.0, because this is something the Ubertooth can easily decrypt and follow. To further inspect what the Ubertooth can and cannot pick up. One thing that can be done is manually setting up a connection between devices and send different kinds of data. During the project this has been done with some standard apps that are used avaiable on the SW2, but with some more control maybe more can be seen.

Another thing that can be investigated is the Smart Connect itself. This is an important app for all Sony devices that you use on mobile devices. The pairing process and the administration of the app's is done here.

References

[1] Get started. `https://developer.sony.com/develop/wearables/smartwatch-2-apis/get-started/`. Accessed: 2014-12-09.

[2] Elaina Chai, Ben Deardorff, and Cathy Wu. 6.858: Hacking bluetooth. 2012.

[3] Mike Ryan. Bluetooth smart. In *Hack In The Box Malaysia*, Malaysia, 2014. HITBOX.

A Pairing

B Ubertooth

C Rogue packets

These are the rogue packets of one packet type for an experiment we have run for 10 minutes.

```
systime=1417701381 ch= 0 LAP=fd9667 err=0 clk100ns=187537993 clk1=105936182 s=-22 n=-87 snr=
Packet decoded with clock 0x40 (rv=1)
  Type: DV/3-DH1
  LT_ADDR: 2
  LLID: 2
  flow: 0
  payload length: 26
  Data:  4f e2 49 69 8b 36 00 45 96 5b 53 91 00 00 00 00 00 00 00 00 00 00 00 00
  Type: DV/3-DH1
  LT_ADDR: 2
  LLID: 2
  flow: 0
  payload length: 26
  Data:  4f e2 49 69 8b 36 00 45 96 5b 53 91 00 00 00 00 00 00 00 00 00 00 00 00
--
systime=1417701487 ch= 0 LAP=fd9667 err=1 clk100ns=1252269813 clk1=106106539 s=-20 n=-87 snr
Packet decoded with clock 0x40 (rv=0)
  Type: DV/3-DH1
  Data:  31 99 cf a9 19 43 80 b3 33
  Type: DV/3-DH1
  Data:  31 99 cf a9 19 43 80 b3 33
systime=1417701487 ch= 0 LAP=fd9667 err=0 clk100ns=1252583583 clk1=106106589 s=-18 n=-86 snr
Packet decoded with clock 0x40 (rv=1)
  Type: NULL
  Type: NULL
--
systime=1417701495 ch= 0 LAP=fd9667 err=0 clk100ns=1323031974 clk1=106117861 s=-18 n=-87 snr
Packet decoded with clock 0x40 (rv=1)
  Type: DV/3-DH1
  LT_ADDR: 2
  LLID: 2
  flow: 0
  payload length: 14
  Data:  95 8a 94 be 3b b2 f4 2a b0 8d ee 1f be cd
  Type: DV/3-DH1
  LT_ADDR: 2
  LLID: 2
  flow: 0
  payload length: 14
  Data:  95 8a 94 be 3b b2 f4 2a b0 8d ee 1f be cd
--
systime=1417701595 ch= 0 LAP=fd9667 err=1 clk100ns=2327602367 clk1=106278592 s=-73 n=-87 snr
Packet decoded with clock 0x40 (rv=0)
  Type: DV/3-DH1
  LT_ADDR: 0
  LLID: 3
  flow: 0
  payload length: 6
  Data:  22 a5 d5 4e fd 97
  Type: DV/3-DH1
  LT_ADDR: 0
  LLID: 3
  flow: 0
  payload length: 6
  Data:  22 a5 d5 4e fd 97
--
systime=1417701650 ch= 0 LAP=fd9667 err=2 clk100ns=2873439317 clk1=106365926 s=-23 n=-87 snr
Packet decoded with clock 0x40 (rv=0)
  Type: DV/3-DH1
  Data:  d4 70 9f f9 25 26 c5 62 d1 7d a3 9b 31 41 7e 08 cb c3 5b 83 f6 64 8f 66 c1 dd a6 da
  Type: DV/3-DH1
  Data:  d4 70 9f f9 25 26 c5 62 d1 7d a3 9b 31 41 7e 08 cb c3 5b 83 f6 64 8f 66 c1 dd a6 da
systime=1417701650 ch= 0 LAP=fd9667 err=2 clk100ns=2873444454 clk1=106365927 s=-17 n=-86 snr
Packet decoded with clock 0x40 (rv=1)
  Type: NULL
  Type: NULL
--
systime=1417701669 ch= 0 LAP=fd9667 err=0 clk100ns=3067100049 clk1=106396912 s=-22 n=-87 snr
Packet decoded with clock 0x40 (rv=0)
  Type: DV/3-DH1
  LT_ADDR: 2
  LLID: 1
  flow: 0
  payload length: 3
  Data:  a7 08 d3
  Type: DV/3-DH1
  LT_ADDR: 2
  LLID: 1
  flow: 0
  payload length: 3
  Data:  a7 08 d3
--
systime=1417701677 ch= 0 LAP=fd9667 err=0 clk100ns=3148338159 clk1=106409910 s=-23 n=-87 snr
```

```
Packet decoded with clock 0x40 (rv=0)
  Type: DV/3-DH1
  LT_ADDR: 2
  LLID: 3
  flow: 0
  payload length: 18
  Data:  07 0e 53 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  Type: DV/3-DH1
  LT_ADDR: 2
  LLID: 3
  flow: 0
  payload length: 18
  Data:  07 0e 53 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
--
systime=1417701757 ch= 0 LAP=fd9667 err=2 clk100ns=667808348 clk1=106537313 s=-23 n=-87 snr=
Packet decoded with clock 0x40 (rv=0)
  Type: DV/3-DH1
  Data:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
  Type: DV/3-DH1
  Data:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
systime=1417701758 ch= 0 LAP=fd9667 err=0 clk100ns=685033527 clk1=106540069 s=-24 n=-87 snr=
Packet decoded with clock 0x40 (rv=1)
  Type: POLL
  Type: POLL
--
systime=1417701795 ch= 0 LAP=fd9667 err=0 clk100ns=1052270825 clk1=106598827 s=-24 n=-87 snr
Packet decoded with clock 0x40 (rv=0)
  Type: DV/3-DH1
  Data:  5f 75 6c 1e cf f1 0f 53 6c d7 5f 88 da 80
  Type: DV/3-DH1
  Data:  5f 75 6c 1e cf f1 0f 53 6c d7 5f 88 da 80
systime=1417701795 ch= 0 LAP=fd9667 err=0 clk100ns=1052275975 clk1=106598828 s=-15 n=-86 snr
Packet decoded with clock 0x40 (rv=2)
  Type: DH1/2-DH1
  LT_ADDR: 2
--
systime=1417701876 ch= 0 LAP=fd9667 err=1 clk100ns=1858355030 clk1=106727801 s=-25 n=-87 snr
Packet decoded with clock 0x40 (rv=1)
  Type: DV/3-DH1
  LT_ADDR: 2
  LLID: 3
  flow: 0
  payload length: 12
  Data:  00 00 00 00 00 00 00 00 00 00 00 00
  Type: DV/3-DH1
  LT_ADDR: 2
  LLID: 3
  flow: 0
  payload length: 12
  Data:  00 00 00 00 00 00 00 00 00 00 00 00
--
systime=1417701995 ch= 0 LAP=fd9667 err=0 clk100ns=3047125143 clk1=106918004 s=-22 n=-87 snr
Packet decoded with clock 0x40 (rv=0)
  Type: DV/3-DH1
  Data:  ce 0c b2 e2 3b d1 cf dc 8a f1 db aa b2 e9 bf 92 ac 69 45 d0 70 66 79 44 92 46 e9 7b
  Type: DV/3-DH1
  Data:  ce 0c b2 e2 3b d1 cf dc 8a f1 db aa b2 e9 bf 92 ac 69 45 d0 70 66 79 44 92 46 e9 7b
systime=1417701995 ch= 0 LAP=fd9667 err=0 clk100ns=3047130291 clk1=106918005 s=-14 n=-87 snr
Failed to decode packet
systime=1417701995 ch= 0 LAP=fd9667 err=1 clk100ns=3049043687 clk1=106918311 s=-70 n=-88 snr
Packet decoded with clock 0x40 (rv=1)
--
systime=1417701997 ch= 0 LAP=fd9667 err=0 clk100ns=3067387404 clk1=106921246 s=-26 n=-87 snr
Packet decoded with clock 0x40 (rv=1)
  Type: DV/3-DH1
  LT_ADDR: 2
  LLID: 3
  flow: 0
  payload length: 34
  Data:  ab 76 62 00 90 91 78 c2 94 b7 18 e5 88 b6 ef 17 7b 3d 14 f5 3e 00 df 7e 00 00 00 00
  Type: DV/3-DH1
  LT_ADDR: 2
  LLID: 3
  flow: 0
  payload length: 34
  Data:  ab 76 62 00 90 91 78 c2 94 b7 18 e5 88 b6 ef 17 7b 3d 14 f5 3e 00 df 7e 00 00 00 00
--
systime=1417702288 ch= 0 LAP=fd9667 err=1 clk100ns=2705025143 clk1=107387556 s=-25 n=-87 snr
Packet decoded with clock 0x40 (rv=0)
  Type: DV/3-DH1
  Data:  b6 88 86 a3 55 97 12 8c 0b 7f 25 ff 25 a4 9a b7 ea 85 90 a5 4e d9 cf 82 13
  Type: DV/3-DH1
  Data:  b6 88 86 a3 55 97 12 8c 0b 7f 25 ff 25 a4 9a b7 ea 85 90 a5 4e d9 cf 82 13
systime=1417702288 ch= 0 LAP=fd9667 err=0 clk100ns=2705162585 clk1=107387578 s=-26 n=-87 snr
Packet decoded with clock 0x40 (rv=1)
  Type: NULL
  Type: NULL
--
systime=1417702348 ch= 0 LAP=fd9667 err=1 clk100ns=23399090 clk1=107482784 s=-29 n=-87 snr=5
Packet decoded with clock 0x40 (rv=1)
  Type: DV/3-DH1
  LT_ADDR: 2
```

```
LLID: 3
flow: 0
payload length: 25
Data:  43 9d 8d cd 3c 5d 8d 2c 46 92 07 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Type: DV/3-DH1
LT_ADDR: 2
LLID: 3
flow: 0
payload length: 25
Data:  43 9d 8d cd 3c 5d 8d 2c 46 92 07 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```