NTNU
Det skapende universitet

Kryptoanalyse og anngrep på Bluetooth

Marius Amund Haugen

Master i kommunikasjonsteknologi
Innlevert: Desember 2012
Hovedveileder: Stig Frode Mjølsnes, ITEM

Norges teknisk-naturvitenskapelige universitet
Institutt for telematikk

Problem Description

Bluetooth Security Attacks and Cryptanalysis

Bluetooth is a personal area wireless communication network intended for secure short-range connections of small mobile devices. This master thesis will investigate the security properties of the Bluetooth channel. This will be done by setting up an experimental lab configuration of Bluetooth device communications, existing eavesdropping hardware devices (Ubertooth), with various antennas, and software tools for collecting and processing intercepted communication. The possibilities of active attacks, such as spoofing and message replay, should also be investigated. The second part of the thesis work will study the reported and published cryptanalytical attacks on Bluetooth, and implement the fastest and most practical cryptanalytical attacks in software, then test the implementation by trying to establish a successful attack on actual intercepted communication from the lab setup. If time allows, a proposal should be worked out for including Bluetooth experiments and cryptanalysis in the lab of TTM4137 Wireless Security master course.

Abstract

Bluetooth is used for short range communication in ad-hoc networks. The technology is used in billions of devices worldwide, and is implemented in a wide range of devices. It connects virtually any electronic device to another without the need for large overhead or complex setup.

The Ubertooth is a Bluetooth monitoring tool aimed at making security assessments available to anyone with the time and interest. It's a fully open source project with an active community. The current state of the Ubertooth provides device discovery of any Bluetooth device, and has the possibility to lock on and follow a Bluetooth conversation. The Ubertooth is not fully developed, but gives monitoring capabilities not present in other openly available tools.

In this thesis, we investigate the usage of the Ubertooth as a tool in attacking Bluetooth. Its capabilities to monitor traffic are tested, and how it can be used in both active and passive attacks is looked at. An injection attack is proposed, but not realized in code. The timing and encoding challenges presented when sending Bluetooth traffic is not yet beaten with the Ubertooth, and it can therefore not be used to send traffic. The attacks presented here is therefor only theoretical, and no implementation is given. The software belonging to the Ubertooth is examined, and the changes necessary to make the injection work is presented. Along with the possibility to use the Ubertooth to enhance existing attacks on Bluetooth.

Acknowledgements

I would like to thank my professor Stig Frode Mjølsnes for support and advice during the work with this thesis. I would also like to thank the people on the Ubertooth General mailing list, and especially Dominic Spill, for helping me and giving me good advice on the Ubertooth.

Contents

List of figures

List of tables

Acronyms

ACO Authentication Ciphering Offset

ACL Asynchronous Connection-Oriented Logical Transport

ACL-U User Asynchronous/Isochronous Logical Link

ACL-C ACL Control Logical Link

CID Channel Identifiers

CRC Cyclic Redundancy Check

DOS Denial of Service

EDR Enhanced Data Rate

FEC Forward Error Correction code

HCI Host Controller Interface

HEC Header Error Check

LAP Lower Address Part

LFSR Linear-Feedback Shift Register

LMP Link Manager Protocol

L2CAP Logical Link Communication and Adaption Protocol

MITM Man-in-the-middle

NAP Non-significant Address Part

PDU Packet Data Unit

UAP Upper Address Part

RF Radio Frequency

SSP Secure Simple Pairing

Chapter 1

Introduction

This thesis is an analysis of how the Ubertooth can be used in attacking Bluetooth communication. When writing this there are only a small number of practical attacks on Bluetooth and only few ways of doing security analysis of a given Bluetooth setup. The goal of the Ubertooth is to bring security analysis on the level of what we see on Wi-Fi today. The main problem of attacking the security on Bluetooth is the fact that commercial Bluetooth adapter's only gives limit information and require knowledge about the devices being monitored. The Ubertooth presents information from all of the Bluetooth channels, and can be used to monitor various Bluetooth devices with relative ease.
The Ubertooth was used in a lab to listen to traffic generated by two commercial Bluetooth devises. The traffic captured gave information about the devices, and about the logical channels used. The payload was however mostly unreadable. Some of the signaling done by Bluetooth is sent in clear text, but most of it is encrypted. Since the encryption is still considered secure there is no way of processing most of the captured traffic. Examining the protocol standard reveals opportunities in the signaling that can be used to attack Bluetooth.
Given that the Ubertooth can be used to observe "hidden" traffic, it gives new possibilities to old attacks and presents new possible attacks. The Ubertooth is however not a finished product, it contains the core functionality you need to monitor Bluetooth traffic, but it lacks functions that makes it possible to send Bluetooth packets. It can send "Bluetooth-like" traffic, but this would not be accepted by a real Bluetooth device due to the way Bluetooth encodes and decodes its packets. Injecting a packet into a live conversation is possible, but still lacks proper implementation to be done in practice. Replaying packets is also possible, but like a crafted packet it requires to be encoded in the right manner to be accepted. Recording and replaying traffic without processing is therefore not an option.
When it comes to cryptanalytic attacks there are none present that can be done

with real traffic. There is one attack that claims to break the cipher used, but the details of how it is accomplished is not disclosed.
As the Ubertooth still lacks some functionality to attack or provide enough information about the traffic captured a lab for TTM4137 is not given here.

Outline

Chapter 2 describes all of the background material used when analyzing the capabilities of the Ubertooth and how it can be used in passive and active attacks on Bluetooth. Chapter 3 described the lab setup used in the analysis. Chapter 4 is a description of how the information obtained by the Ubertooth can be used in attacking Bluetooth. Chapter 5 is the analysis of the possibilities the Ubertooth presents and describes the weaknesses it can exploit. This also describes the limitation of the Ubertooth. Chapter 6 is the final conclusion and describes what the author thinks can be done with the Ubertooth given further development.

Chapter 2

Background

In the first section of this chapter the Bluetooth architecture and its security measures relevant to the lab are described. In the second section known weaknesses in Bluetooth is presented. The third presents an overview of the Ubertooth and is capabilities.

2.1 Bluetooth

Bluetooth is a short range Radio Frequency (RF) technology that operates in the 2.4 GHz band, connecting various devices together in a ad-hoc manner. Bluetooth enabled devices span from mobile phones, PCs, headsets to USB-adapters. On a worldwide basis the number of devices number in the billions [1]. At the time of writing this, there were 4 core versions of Bluetooth. The material presented here will be from version 2.1 due to the hardware used in the lab.

2.1.1 The Bluetooth Architectural Overview

Topology

Bluetooth uses a master slave topology that is defined ad-hoc. When devices connect to each other they form a piconet consisting of one master and one or more slaves.

A device can only be master of one piconet at a time, but it can be slaves in several. In Figure 2.1 A and E are masters, D is a slave connected to two piconets.

Protocol stack

The protocol stack can be divided into two main parts, controller and host. The host has the responsibility of the upper levels, containing resource and channel management. The controller handles the lower levels, containing link management, link control, baseband resource management, and the physical layer.

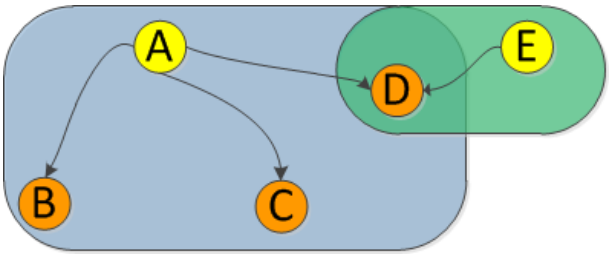

Figure 2.1: Example of two piconets

Figure 2.2 is a simplified description of the Bluetooth stack, containing only the main components.

A common setup is with a device with computational power (e.g. a PC) that implements the host part, with an adapter that takes care of the controller and RF part. This kind of setup uses a Host Controller Interface (HCI) to communicate between the two components.
To make the setup of Bluetooth easier to explain the rest of this thesis will focus on the two main parts in Figure 2.2, mainly the Logical Link Communication and Adaption Protocol (L2CAP) layer and the Link Manager Protocol (LMP). For simplicity the HCI will be ignored, as this is not present in all types of implementations, and not vital for the explanations done here.

Addresses

Bluetooth addresses are a 48-bit unique identifier divided into three parts:

- Lower Address Part (LAP), consisting of 24 bits
- Upper Address Part (UAP), consisting of 8 bit
- Non-significant Address Part (NAP), consisting of 16 bit

The address may take any value except from 64 reserved LAP values used for inquiries. The general inquiry LAP is 0x9E8B33, this is used by all devices to send inquiry requests.

Logical channels

Bluetooth uses Asynchronous Connection-Oriented Logical Transport (ACL) channels to send data between devices. There are two channels used for basic rate transmissions.
The ACL Control Logical Link (ACL-C) carries LMP Packet Data Units (PDUs).

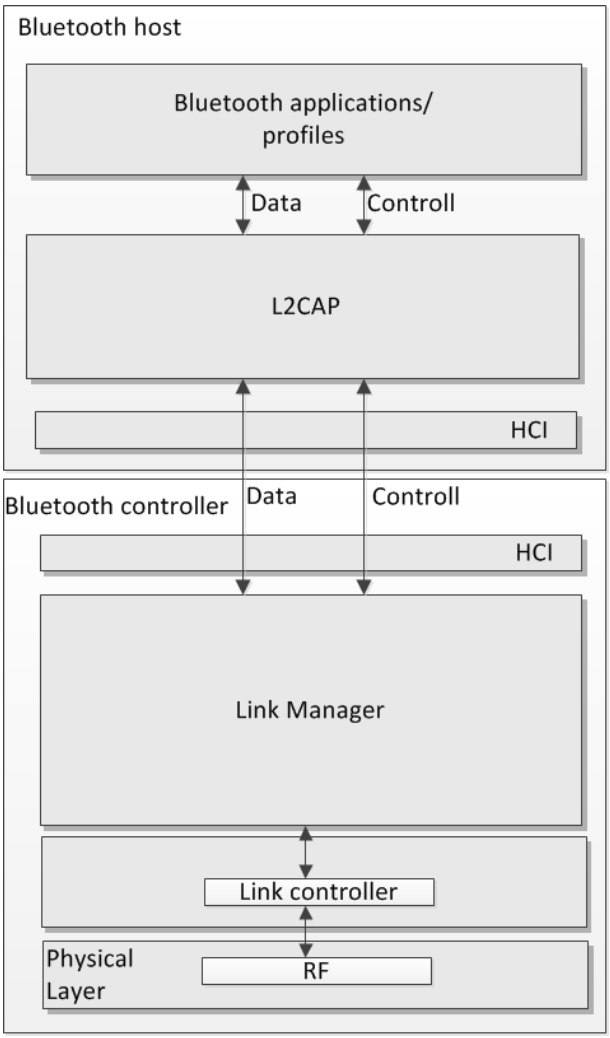

Figure 2.2: The Bluetooth stack. Modified from [2]

The User Asynchronous/Isochronous Logical Link (ACL-U) carries L2CAP packets, this can either be user data or L2CAP PDUs.

Bit processing

The last processing a packet goes through before its sent out on the RF channel is the whitening and Forward Error Correction code (FEC) encoding. The header is always FEC encoded, and the payload of some packets are. The whitening is done by XOR-ing the data with a whitening word. The whitening word is obtained by using the master clock as input to a Linear-Feedback Shift Register (LFSR). This must not be confused as a security measure, but a RF technique used to improve transmissions through noise.

The FEC encoding is done at a rate of 1/3 or 2/3, dependent on the type of packet. This is done to reduce the number of errors caused in transmission.
A FEC with a rate of 1/3 will repeat every bit three times. This will make the sequence ”1010” into ”111000111000”.
A FEC with a rate of 2/3 is a (15,10) shortened Hamming code. This is a LFSR generating code that will produce 15 bit for every 10 it receives.

Hopping sequence

Bluetooth divides the 2.4 GHz band into 79 different channels; these channels are used in a pseudo random hop sequence. The hopping sequence for the piconet is determined from the address and clock of the master device. The hopping sequence therefore determines the physical channel a piconet uses, and limits a device to being a master in one piconet at time. A Bluetooth device will hop at rate of 1600 times per second.

Data rates

Basic data rate and Enhanced Data Rate (EDR) transmissions uses different packet format, but the access code and header are identical for both packets. EDR packet has a guard space, sync word and a trailer in addition to the payload.

Class of devices

There are three classes of Bluetooth devices. Class 1 has a maximum range of 100 meters, class 2 has a range of 10 meters and class 3's range is 1 meter [3].

2.1.2 Asynchronous Connection-Oriented Logical Transport

The ACL is the the data carrier for the logical channels using basic rate[1]. It defines the types of packets that are used and separates the logical channels by a payload header. Both channels use the same basic packet format, but the type of packet and the type of payload differs between the channels.

Packet format

The general packet format for basic rate packages consists of three parts: access code, header and payload. Figure 2.3 describes the complete packet, with the values contained in the access code and the header.

The access code is 72 bit identifier for the piconet. It consists of three fields:

- Preamble, 4 bit

[1] This applies for both ACL-U and ACL-C

- Sync word, 64 bit
- Trailer, 4 bit

Both the preamble and the trailer consist of alternating ones and zeros[2]. The sync word is derived from the LAP of the master.

The header is 54 bit[3] and consists of six fields:

- LT_ADDR, 3 bit logical transport address
- Type, 4 bit packet type identifier
- Flow, 1 bit flow control
- ARNQ, 1 bit acknowledge indicator
- SEQN, 1 bit sequence number
- HEC, 8 bit error check

The LT_ADDR is a logical address assigned by the master to the slaves in the piconet. This is used to identify the slave a packet is going to or from. The master has no logical address; the direction of the packet is determined by the timing of when the packet is sent.
The flow bit is used to stop or start the traffic on the ACL link.
The sequence number is used together with the acknowledgment bit to provide reliable transport of some types of packets. The Header Error Check (HEC) is a sum that is calculated to check that the header does not contain errors.

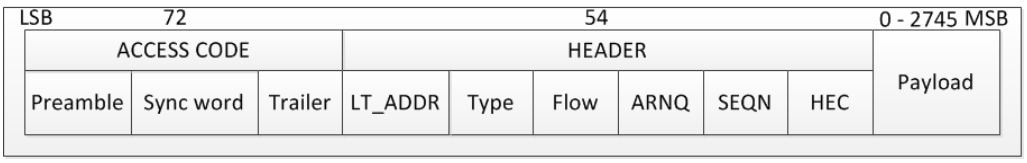

Figure 2.3: Basic Rate packet structure.

The payload is 0 to 2745 bit long and can contain any type of data. Some packet types append a Cyclic Redundancy Check (CRC) to the end of the payload field as a guard against errors. Figure 2.4 show an example of a payload containing a PDU, note that the CRC is not appended all types of packets.

[2]The sequence is either 1010 or 0101 depending on the adjacent bit. The last bit of the preamble is always the opposite of the first bit in the sync word. The first bit of the trailer is the opposite of the last in the sync word.

[3]The header in its raw format only spans 18 bits, but is expanded to 54 bits when it is encoded.

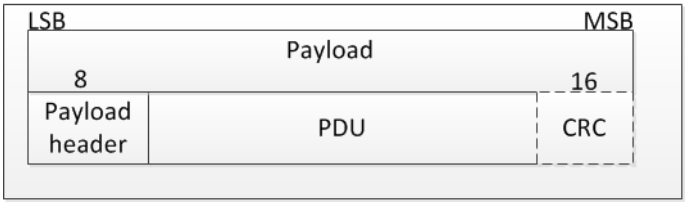

Figure 2.4: A PDU as payload

Payload Header

The ACL packets has a payload header that defines what type of logical link it belongs to. The ACL-U is used for L2CAP messages. The ACL-C is used for LMP messages. The header for basic rate packets and un-fragmented packets consists of three fields.

- LLID: Two bit that indicates ACL-U message, continuation of a ACL-U message or ACL-C message[4].
- Flow: One bit that stops or starts the traffic on L2CAP level.
- Length: Five bit, contains the length of the payload that follows.

For all EDR packets and ACL packet that are fragmented into several packets, the payload header is 2 bytes. The difference from basic rate header being a length field of 10 bits and 3 reserved bits (set to 0) at the end. Figure 2.5 shows the palyload header for a ACL packet sent with basic rate.

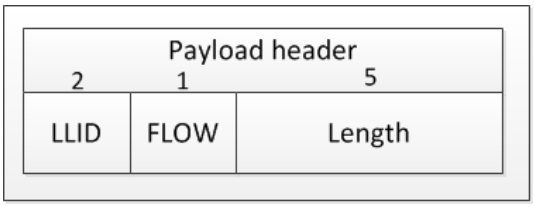

Figure 2.5: ACL payload header

Packet types

ACL carries eleven different packets. Four of the are special control packets that is only used in specific cases. The packets and their main properties are listed underneath:

[4]01 indicates continuation of ACL-U, 10 is start of a ACL-U. 11 indicated a ACL-C

Type	Payload header size in bytes	User payload in bytes	FEC	CRC
DM1	1	0-17	2/3	yes
DM3	2	0-121	2/3	yes
DM5	2	0-244	2/3	yes
DH1	1	0-27	no	yes
DH3	2	0-183	no	yes
DH5	2	0-399	no	yes
AUX1	1	0-29	no	no

Table 2.1: ACL packet types

- ID: Identity packet only containing device access code, derived from the address of the paged device.
- NULL: Contains only channel access code and packet header.
- POLL: Similar to the NULL packet. Can only be sent from the master and has to be acknowledged.
- FHS: Is a special control packet used to synchronize the hopping sequence. This is only used in page, inquiry and role switch. This packet is FEC 2/3 encoded and has a CRC.

The FHS packet is a Frequency Hopping Synchronization packet, that contains 144 bits of information. The most important fields are: The entire address of the sending device, the class of device, LT_ADDR, and a 26 bit native time value. This is enough information so that the receiving side can determine the hopping sequence the sending side is following.

The ACL also carries general packet types used for both user data and control messages. The packet types listed in Table 2.1 is all of the packets that are sent over ACL with basic rate, with their main difference provided in the table.

2.1.3 Link Manager Protocol

This controls all of the link layer operations. This includes link establishment, pairing and encrypting the link. This is at this layer the packages are formatted before transmission, checksums are calculated, and encryption/decryption is handled. To communicate between link managers in a piconet predefined PDUs are used.

LMP PDU

All LMP PDUs are send with DM1 as packet type. All of the PDUs are predefined and can only hold specific values. The PDU hold three fields:

- Transaction ID, a one bit direction identifier
- OpCode, either 7 or 15 bit PDU identifier
- Payload, dependent on on the type of OpCode used.

Figure 2.6 shows the general format of a LMP PDU.

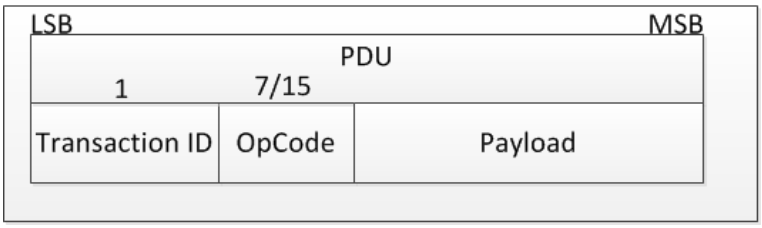

Figure 2.6: LMP PDU format

It's possible to fragment PDUs over several packets, and each of the fragments has to be acknowledged[5] by the recipient before the next fragment is sent. Because of this there is a possibility for a collision if a device needs to send a command to adjust some of the link properties. There is therefore defined a set of PDUs that always will be accepted. The PDUs is listed below.

- LMP_channel_classification
- LMP_decr_power_req
- LMP_incr_power_req
- LMP_max_power
- LMP_max_slot
- LMP_min_power
- LMP_preferred_rate
- LMP_set_AFH

[5]This i done with a LMP_accepted PDU

Connection procedure

When setting up a Bluetooth channel with version 2.1 or greater, devices follows a number of steps that insures that the channel is secure and the connection is to the right devices. First the link is established. Figure 2.7 sums up the link establishment procedure, the parentheses indicates what kind of packet that are used.
The procedure starts with one of the devices, the non-initiating part, in discoverable mode, which means that it listens for inquiry messages. The initiating device sends out an inquiry message on a predefined channel common to all devices. The non-initiating device responds with a FHS packet. The initiating device will be the master of the piconet, and the responding will be a slave.
Knowing the address and the time of the other device the master sends a paging request on a channel specific to the slave. Upon receiving a ID packet as a response on the same channel, the master send a FHS packet. This is acknowledged by a ID packet by the slave. The slave now knows the hopping sequence of the piconet, and will start listening accordingly. The master confirms this by sending a POLL to the slave.
The master sends a feature request to the slave, the response to this message contains capabilities used to determine the level of security used during pairing. The master then sends a connection request, and upon receiving a response paring is initiated.

Secure Simple Pairing

The pairing and authentication procedure is called Secure Simple Pairing (SSP), and is used on all devices using version 2.1 or higher. When a link is established this is used to to create a secure Bluetooth. All of the messages sent are in clear text until the channel establishment procedure is complete and the encryption is switched on.
The first step of pairing is to exchange IO capabilities. This is done via a LMP PDU and decides what kind of association model that is used. The next step is then to exchange public keys between the devices. The first step of authentication then follows. This uses one of four association models: Numeric Comparison, Just Works, Out-of-Band or Passkey Entry.

Numeric Comparison will be used when both devices has a display that can display six digits and both of them are capable of answering ”yes” or ”no”. The user has to confirm that both devices are showing the same number. An example is between a PC and a phone.

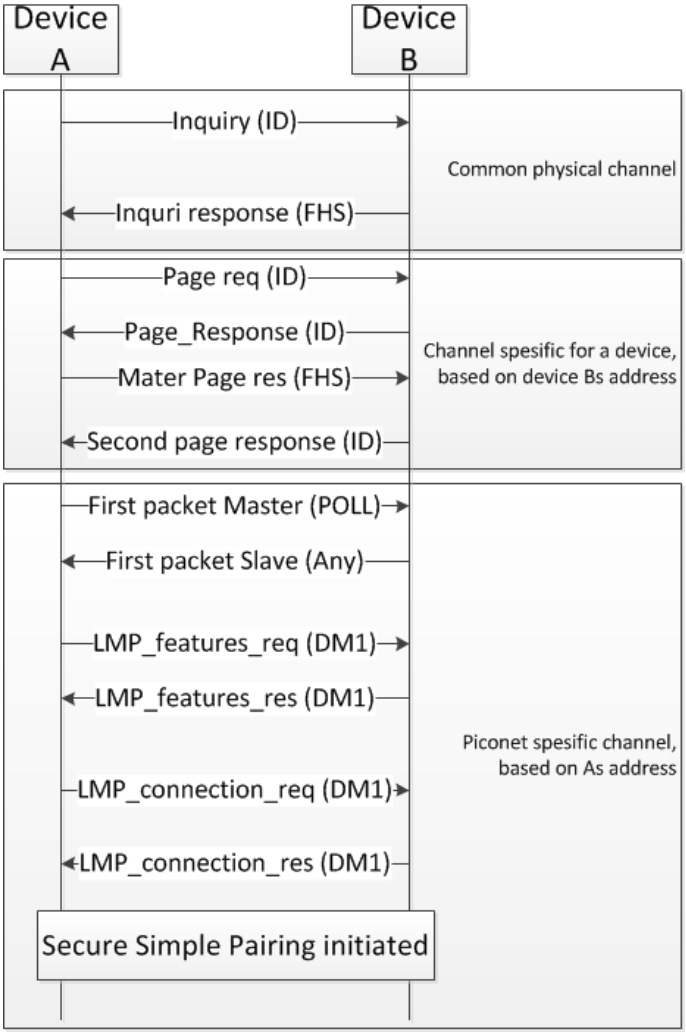

Figure 2.7: Link setup

Just Works is used when one of the devises cannot display six digits or does not have a keyboard to enter six digits. The user may just be asked to accept the connection. An example is a phone and headset.

Out of Band is designed for the scenarios where an out of band mechanism is used for both device discovery and cryptographic exchange.

Passkey Entry is used when one of the devices does not have a display, but has input capabilities, and the other device has a display. An example is a PC and a keyboard.

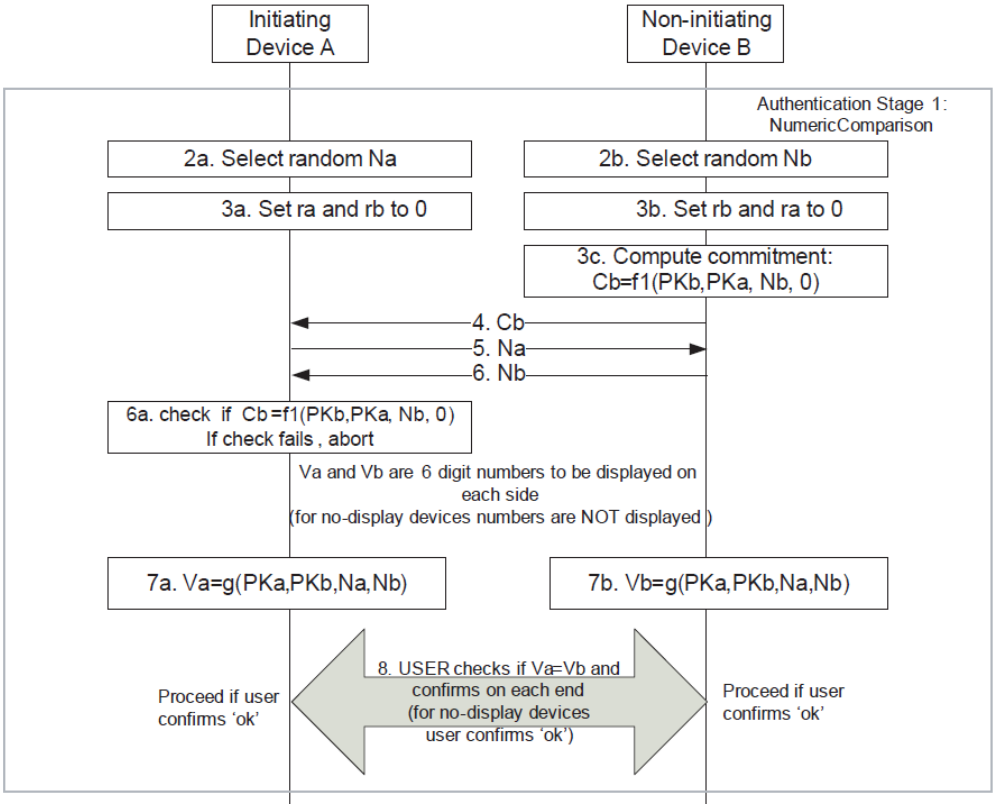

Figure 2.8: Authentication Stage 1 with Numeric Comparison. Taken from [4].

Figure 2.8 describes the first authentication step when Numeric Comparison is used.

When authenticated, both devices calculate a confirmation value containing a shared key derived in the first part of the authentication process. This is sent to the other device and confirmed. If there is an error in the conformation value the link will be terminated. Upon confirmation both sides calculate a link key. When this is done the devices is considered paired. Figure 2.9 describes the calculation of the confirmation value and the link key. Note that the IO capabilities used in the figure comes the PDU exchanged at the start of the paring.

2.1.4 Logical Link Communication and Adaption Protocol

At the L2CAP layer data received from applications are broken down into smaller chunks for processing on the underlying layers. The L2CAP layer also has the responsibility of maintaining Bluetooth channels, as well as quality of service settings for the channel. To transmit data L2CAP uses the ACL-U and uses its

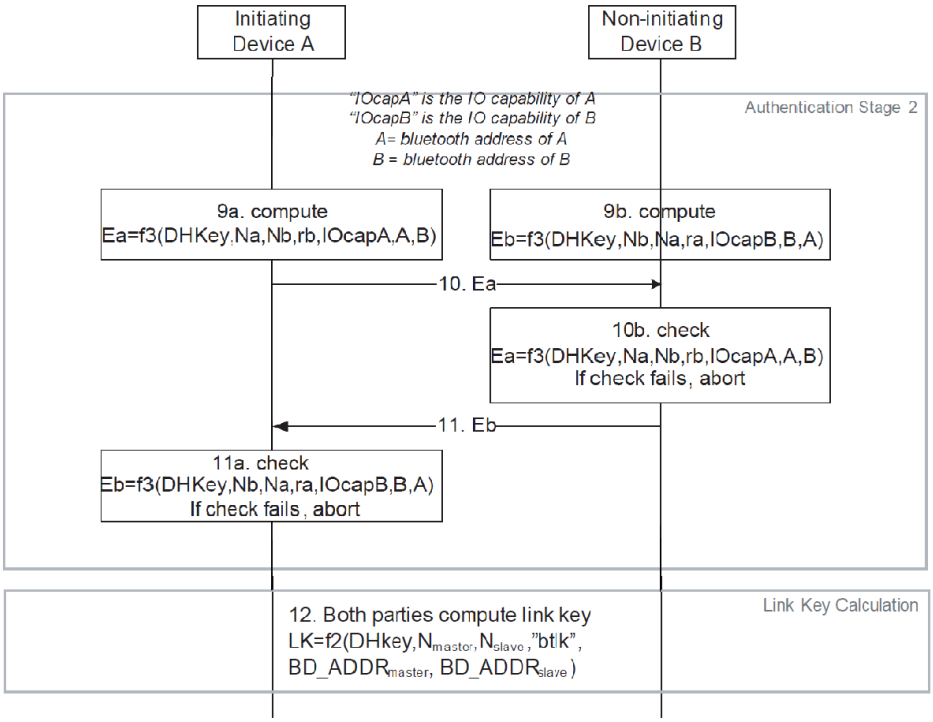

Figure 2.9: Authentication Stage 2. Reproduced from [4].

own PDUs to contain the information.

PDUs

The L2CAP PDUs consists of two fields: a header and a payload of variable length. The header contains the length and the channel id and is present in all L2CAP PDUs
The rest of the PDU is dependent of what kind of information that is carried. The types of information are separated into frames. Figure 2.10 is an example on a L2CAP PDU with a C-frame.

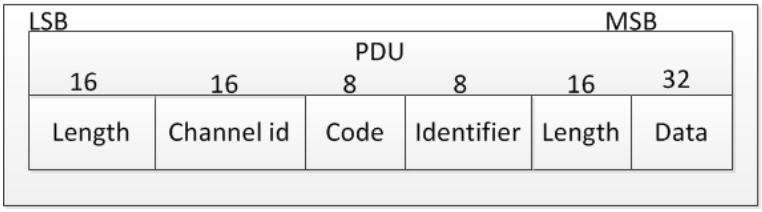

Figure 2.10: Example of a L2CAP C-frame PDU

- B-frame: Basic information frame, contains only an information payload.
- I-frame: Information frame, contains a control field, length field[6], payload and a Frame Check Sequence[7].
- S-frame: Supervisory frame, contains only a control field and a Frame Check Sequence.
- C-frame: Control frame, contains predefined commands as payload.
- G-frame: Group frame, contains a Protocol/Service Multiplexer field, and payload.

PDU header

The L2CAP header contains a length field that defines the length of the payload in the PDU, and a Channel Identifiers (CID).
L2CAP uses its own ID's to separate the channels in a piconet from each other. Upon channel establishment the master sets a CID that defines the endpoints. This makes it possible to tell the difference of the traffic sent on different channels and the direction of it. Some of the name space are reserved, but most of it is dynamically assigned and can take a value between 0x0040-0xFFFF. The ID is present in all PDU headers and defines the endpoint the packet is heading for. In some signaling packet there are two IDs, for both sender and receiver[8].

L2CAP has its own signaling packets it uses for channel management. These packet are sent in a C-frame with CID = 0x0001, a command code, a identifier, length field and a data field.
The command code used in the signaling packet is twelve predefined values. Table 2.2 sums up the signaling commands and its description. The identifier is a way of tracking requests and responses. A new request will take the next available value, and the response will contain the same value. When the available values are used up it starts with the first value again.
The length field here is the length of the data field that follows. The data field contains different values based on what code that is used. Common values are CIDs or data relating to a request.

[6] Used when the payload is fragmented into several frames
[7] Used for error detection
[8] Named DCID for Destination CID and, SCID for Sender CID

Code	Description
0x00	Reserved
0x01	Command reject
0x02	Connection request
0x03	Connection response
0x04	Configure request
0x05	Configure resonse
0x06	Disconnection request
0x07	Disconnection response
0x08	Echo request
0x09	Echo response
0x0A	Information request
0x0B	Information response

Table 2.2: Signaling Command Codes

2.1.5 Bluetooth Security Measures

Discoverable modes

A Bluetooth device can be in one of three discoverable modes: Either non-discoverable, limited discoverable or general discoverable mode. The discoverable modes only restricts inquiry messages, paired devices can be non-discoverable and still communicate.

- Non-discoverable mode: A device in this mode will never answer on an inquiry message.
- Limited discoverable mode: A device in this mode will answer on inquiry messages for a limited amount of time.
- General discoverable mode: A device in this mode will answer on all inquiry messages it receives.

Connectable modes

A Bluetooth device can be in either connectable mode or in non-connectable mode. When it is in non-connectable mode it will never answer to any paging requests. When in connectable mode it will answer to any paging request it receives.

Bondable modes

There are two bondable modes, non-bondable and bondable. Non-bondable will refuse any pairing requests, bondable will accept them.

Security modes

Bluetooth version 2.1 uses security mode 4 when establishing a channel. This is a service level security that uses the highest level of security available at the remote device. When connecting to a 2.1 device Simple Secure Pairing is required. A device in security mode 4 will however respond to devices in security mode 3 due to backwards compatibility. When connecting to devices with a version pre 2.1 security mode 2 is used. The full list of security modes and its requirement is as follows:

- Mode 1: No security.
- Mode 2: Service level enforced security. Authorization, Authentication and Encryption are required. Security enabled after channels is established.
- Mode 3: Link level enforced security. Security will be initiated before channel setup is completed. A device can reject connection requests.
- Mode 4: Service level enforced security. Requires either an authenticated link key, an unauthenticated link key or no security[9].

Keys and encryption

Before any encryption can occur a common encryption key has to be created. This is done after the link is authenticated and Secure Simple Paring has been complete.
When establishing a secure channel a public-private Elliptic Curve Diffie-Hellman key pair is used for the initial stages. This key is generated on the device in the initial stages of pairing and can be discarded at any time. From the public key of the other device a Diffi-Hellman key is calculated. After one of the four association models is used to authenticate, the Diffi-Hellman key is used to confirm that the pairing was successful through a challenge-response. This is what was described in Figure 2.9 under Authentication Stage 2.

The keys that are used and how they are created is as follows:
Initialization key, K_{init}, is generated using a Bluetooth address, a PIN[10] and a random number. Before authentication is achieved this is the link key
Authentication is achieved by a challenge-response scheme. It uses a random number, the Bluetooth address of the device being challenged, and the current

[9] An authenticated link key is one there either out of band, numeric comparison or passkey entry was used during simple secure pairing. An unauthenticated link key is one where just works is used.

[10] The PIN used by one of the devices can be fixed, but not both. The PIN can be input by a user or generated at the application layer

link key. A successful authentication generates a Authentication Ciphering Offset (ACO).
Combination key is calculated by two random numbers, LK_K_A and LK_K_B, one from each device. LK_K_A and LK_K_B is calculated by the devices own address and a random number. The number is then transferred to the other device by XOR-ing it with the current link key. A new link key, K_{AB}, is then derived by XOR-ing LK_K_A and LK_K_B. Figure 2.11 describes the generation of the combination key.
Encryption key, K_c, is derived from a random rumber, ACO, and the link key.

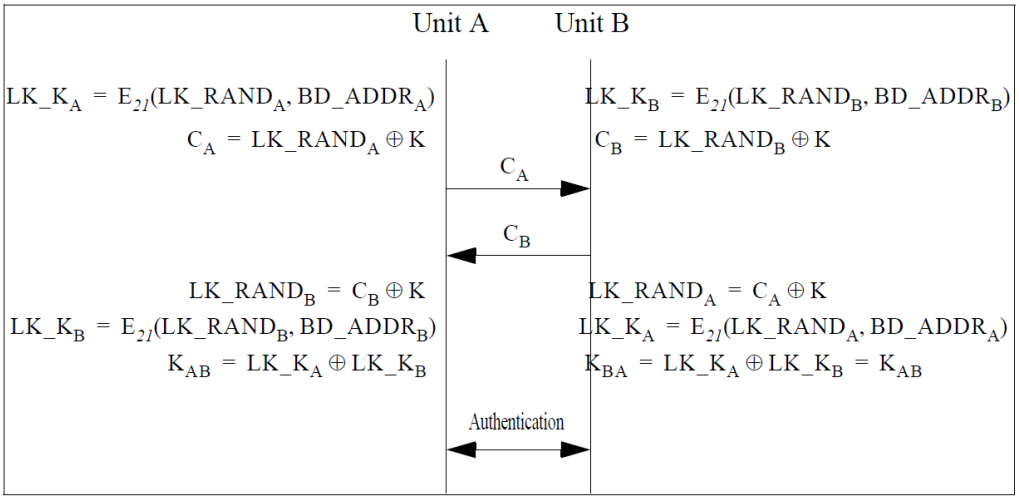

Figure 2.11: Encryption procedure. Taken from [4]

Note that the random number used here is chosen by the master, and transmitted to the slave in clear text. This is denoted EN_$RAND_A$ in Figure 2.12
Encryption of the payload in packets are realized by using K_c, address to the master device, the native time value of the master[11]. This is a stream cipher that updated for every packet that is sent.
Figure 2.12 describes the encrypt and decrypt procedure.

After the Encryption Key has been created the encryption can be turned on. Encryption is only used on the payload of packets and are controlled by the LMP.

Secure Simple Pairing

Secure simple pairing gives a number of security measures that protects the user. The link key and the encryption used by Bluetooth gives a strong protection

[11] This is sent in the FHS packet at link establishment

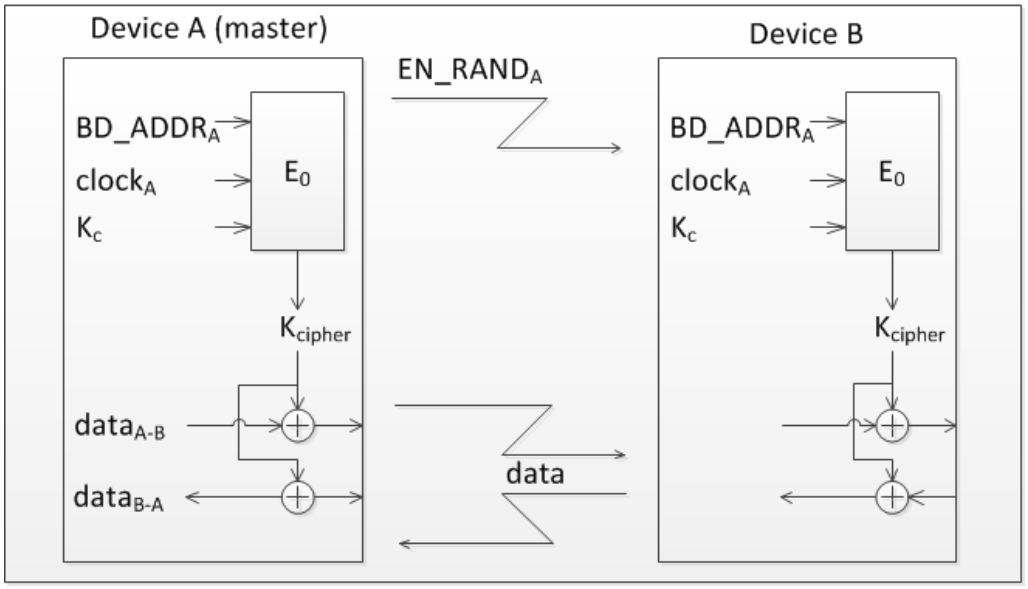

Figure 2.12: Encryption procedure. Taken from [4]

against passive eavesdropping. The values used to derive the link key is protected by a private-public key pair during set up, making inserting false values almost impossible. Man-in-the-middle (MITM) protection is provided by the Numeric Comparison and Passkey Entry association models. For an attacker to be successful, the 6 digits used in these models have to be guessed.

2.2 Attacks

2.2.1 Active

There are a number of proven active attacks on Bluetooth. Some of them are attacks on mobile phones, exploiting flaws in the manufacture implementation and user settings [5]. There are a range of Man-in-the-middle attacks using USB-adapters with changeable Bluetooth addresses [6, 7].

Man-in-the-middle

To execute a successfull MITM attack you need to impersonate two legit devices. One can achieve this by changing the addresses on USB-adapters to the addresses of the devices being attacked. Re-programmable USB-adapter is available and software used to re-write firmware can be found online [6]. These types of attacks rely on that you can get the address of the target devices, and force a reconnect. By timing it right it is possible get the targeted devices to connect to

the modified adapters in stead of the real ones. By resending the traffic from both the targeted devices the connection will appear legit. Since the attacker now sits in between the information is not encrypted and can be read as clear text.
Figure 2.13 describes the general nature of MITM attack. This case forces the targeted devices to use Just Works association model upon connection. This is done by pretending to have no capability to see or enter a code. By doing this the MITM protection provided by the other association models is avoided.

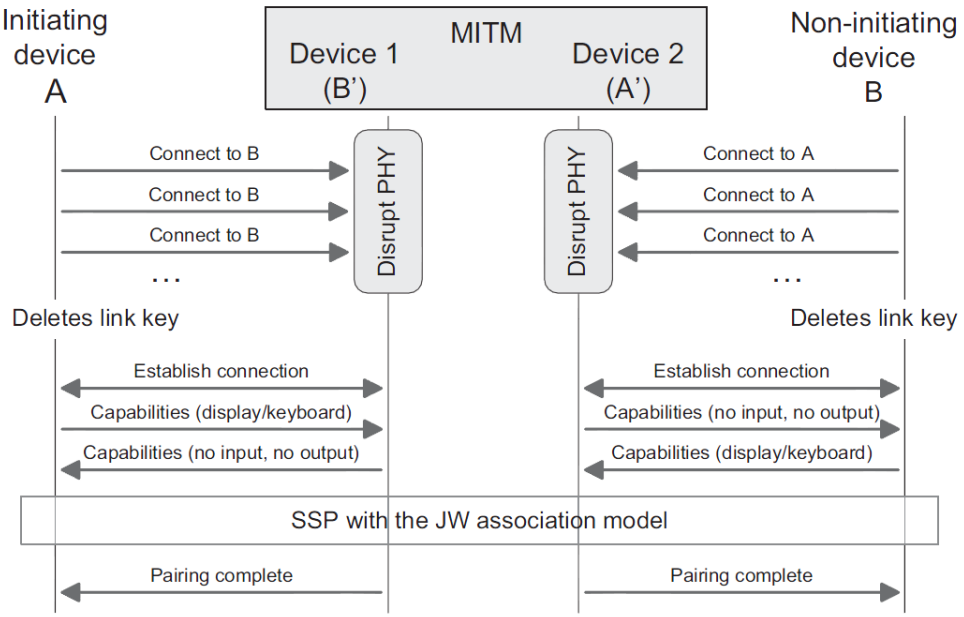

Figure 2.13: Man in the middle. Taken from [6]

Attacks on mobile phones

Bluesnarf and Bluesnarf++ is attacks that give the attacker access to the file system of a phone. This exploits a flaw in the OBEX implementation that grants remote access without pairing.
BlueBug is a vulnerability present on some mobile phones that lets an attacker execute AT commands that takes over the control of the device.
Bluejack lets a attacker send anonymous vCards or text messages to a phone. Physically harmless.
HeloMoto a combination of Bluesnarf and Bluejack. Only works on some Motorola phones.

2.2.2 Passive

Passive attacks are an attack that can be performed without sending any traffic to the target. Similar attacks on Wi-Fi are performed by recording data traffic first, and use the collected data to break the crypto used. Another type of passive attack is to use device addresses to provide location tracking for specific devices.

Key stream attack

In the authors knowledge there are only one practical attack on the stream cipher used by Bluetooth. In [8] a zero-knowledge-like analysis of E0 is presented. Zero-knowledge-like meaning that the basis for the analysis and its computation is given, but not the weaknesses.

The E0 chipher is a stream cipher that produces a random output sequence given that the keys used are random. The paper goes out to find keys that produces an output with a given property and targeted Hamming weight, thus proving that the cipher has weaknesses. The length of the input used in the proof is n=128 bit. This length is chosen to resemble the output used in real life implementations of Bluetooth, and would be usable in a real attack.

The first property used in the cryptanalysis is Hamming weight[12]. The goal is to find keys K such that a 128 output sequence has a Hamming weight (at most) equal to a value k. The second property is a number of consecutive zeros r that appear anywhere in the stream.
The interesting thing here is the computational complexity of finding such a sequence. The computational complexity of finding a given Hamming weight at least k, by random search with n = 128, is denoted C_k. C_r is the complexity of finding a sequence of r consecutive zeros that appear at the start of the stream. C_{r+} is consecutive zeros anywhere in the stream. The complexity of finding a sequence cumulating both values k, and r is denoted $C_{r,k}$, and similar with r+ and k is denoted $C_{r+,k}$. Table 2.3 sums up some of the values presented in the paper, if you are to use a random search to get a sequence that exhibits that kind of properties.

The weaknesses apparently present in E0 is detected by the use of two non-public packets used in a pre-computing step, namely CoHS[3] and VAUBAN. The weaknesses are combinatorial in nature, and are to the authors' knowledge not published. The pre-computing step is said to take one week to be performed, but is only required to be done once. With the use of this information different values for r and k is a cryptanalysis step is preformed and the first keys is re-

[12]The total number of non-zero bits in a sequence

(r,k)	C_k	C_r	$C_{r,k}$	$C_{r+,k}$
(69, 29)	$2^{32.76}$	2^{69}	$2^{72.28}$	$2^{66.40}$
(69, 27)	$2^{36.39}$	2^{69}	$2^{72.57}$	$2^{66.69}$
(69, 25)	$2^{40.30}$	2^{69}	$2^{73.25}$	$2^{67.36}$

Table 2.3: Complexity Comparison for Hamming Weight and Run Properties (Random Search; n = 128) Modified from [8]

trieved within the first hour. In a period of five weeks it is claimed that 48,000 keys is retrieved.
The paper gives a number of keys that has been retrieved by their use of the weakness. Where one of the most important keys produce a output that has a hamming weight of 29 and 69 consecutive zeros at the start. To obtain a key that can produce such a sequence by random search has a complexity of $2^{72.28}$. This is not something that can be achieved with existing computing resources. The overall complexity of this attack, given the time used and keys obtained is calculated to be 2^{35}. Compared with other known attacks on E0, with the same size known key stream bits, the best is 2^{86}[9]. This attack also uses assumptions that would make it unusable in a real attack.

This paper gives the reader information to verify the findings, but does not disclose any information about which weaknesses that is exploited. This is done for several reasons. If a paper discloses detailed information on how to break an encryption, it will be available to anyone that wishes to exploit it. It will apply pressure to change the encryption, but that is not something you can do over night. Vendors will have financial reasons to delay a switch of technology, and users are in general reluctant to change equipment. The time spent to change out all of the affected devices, with Bluetooth devices ranging in the billions, would probably be counted in years.

Location tracking

Location tracking is not a traditional attack, but it detects the possible presence of a person by detecting a Bluetooth device [10]. It does no harm to the device or the traffic transmitted, but it poses as a privacy concern. The addresses used in Bluetooth are unique and are set by the manufacturer of the device; it can therefore be used as a mean to identification. There is possible to alter the addresses of some devices, but in the author's opinion, the only reason to do this is to perform an attack, and will not be something a normal user would do.
In a normal situation a user would be protected against this as long as the Blue-

tooth device is set to non-discoverable. Even if the address is known an attacker would not be able to confirm the presence of a device without being paired with it. This makes the privacy relatively secure.

2.3 Ubertooth

The Ubertooth is a 2.4GHz transceiver built specifically to monitor and inject Bluetooth traffic. The Ubertooth project is open source, aimed at making Bluetooth security analysis available to anyone. All of the hardware plans and software needed to make one is available on the projects website [11].

2.3.1 History

Project Ubertooth is the product of Michael Ossmann's quest of trying to do Bluetooth security assessment easier. The result is Ubertooth Zero and Ubertooth One, both are USB-adapters available through his company Great Scott Gadgets [12]. For more information on the development process see his presentation from ShmooCon 2011 [13].

2.3.2 Specification and Capabilities

At the time of writing the Ubertooth is capable of Basic Rate Bluetooth monitoring, including LAP and UAP discovery, hop sequence discovery and following a given piconet. It is also capable of Bluetooth Low Energy monitoring[13]. This section will only focus on the main capabilities used in the lab, the full list of tools is briefly listed in Chapter 3.

Address discovery

One of the Ubertooth's functions is address discovery, and it can be achieved in two different ways. The first is the use of the ubertooth-lap and ubertooth-uap tools. The -lap will look for devices and recover the LAP for any device in range, but only one at the time. The -uap requires the LAP of a device and will recover the UAP for that specific device.
The second method is by the use of the Kismet[15] plugin, and once installed it is the most user friendly. The plugin lists all of the Bluetooth addresses the Ubertooth is picking up, and the number of packets associated with the address. A note here is that the Ubertooth produces a number of false positives, an address observed only a couple of times is most likely a packet decoded wrong. There is also a difference between the packets that is recognized with both LAP and UAP

[13]Bluetooth low energy is a standard used for devices intended to operate for extended periods of time on batteries [14]

or just the LAP. Bluetooth uses a specific and shared address for device discovery, meaning that there will be an address listed with a number of packets that originates from any device. Since this is a fixed address it is easily recognized, and can be ignored, though it is an indication that there are devices actively trying to discover other devices.

Ubertooth-hop

This tool is essential when trying to monitor a specific piconet. The pseudo - random hopping sequence used by Bluetooth means that the traffic in a piconet will be obscured from any observer, unless one can monitor all of the 79 channels at the same time or follow the hopping sequence. Ubertooth-hop requires the UAP and the LAP of one of the devises in the piconet, and will stay on one of the channels observing packets and try to calculate the hopping sequence by determining the clock of the master. Once the sequence is discovered the Ubertooth can begin to hop along with the piconet and observe the packet stream. Once in hop-mode the Ubertooth begins to try to decode the packets observed. A successful decode means that the packet is unwhiten, the FEC and the HEC is reversed, and the CRC-check is correct. The tool will print out information from the header and any payload present. Note that the payload is still encrypted. There is no guarantee that the received packets are decoded correctly.

The Ubertooth does not support EDR, so it can only be used to partially observe EDR transmission. Meaning that is can still recognize the access code and the header, but not the payload. Signaling packet however is sent with basic rate, and is therefore fully observably.

2.4 Other Bluetooth Sniffers

There are professional Bluetooth sniffers available for developers of Bluetooth devices. These sniffers have complete control over any and all targeted Bluetooth traffic.

2.4.1 Capabilities

The professional sniffers are developer tools used to make a Bluetooth implementation work as it is intended to do. To do this full control over any message that is sent, on any level, is needed. The site [16] provides such a sniffer.
The sniffers provided there displays all types of messages from HCI to RF traffic, and they can even decrypt encrypted packets. This is used to debug any fault in the messaging.

2.4.2 Limitations

There are however major limitation to the use of such sniffers. Since these are intended for developers there are prerequisites for using its full potentials.
For one, the addresses and time value for the devices targeted are required to be able to follow a piconet from its formed.
To decode the encrypted traffic the link key for the devices has to be entered into the sniffer.
These are things you will be able to get if you have full access to the devices used. As an attacker using these devices you will probably have none of it.

Chapter 3

Lab

This chapter explains the lab setup used to examine the use of the Ubertooth in this thesis. You will also find the measurements data from range testing in this chapter.

3.1 Hardware

The hardware used in the lab was two computers, two Bluetooth 2.1 USB-adapters, one HTC Sensation, and one Ubertooth One.

3.1.1 Bluetooth Controllers

When generating traffic to observe with the Ubertooth two computers and a mobile phone was used.
Computer-1 was running Windows 7, and was used to generate Bluetooth transmissions, either with the second computer or the mobile phone. Computer-2 was running Ubuntu 12.04 and was to generate Bluetooth traffic, and as the Ubertooth controller. The main reason for running Ubuntu on this machine was because of the requirements for installing and running the Ubertooth software. The mobile phone was only used to receive traffic at a distance from the Ubertooth to collect range data.

3.1.2 Bluetooth Adapters

The two USB-adapters were designated Alice and Bob. The adapters was identical, they were class 1 devices and was using Bluetooth version 2.1 with EDR.
Alice: Used with Computer-1. Bluetooth address: 00:15:83:43:60:25
Bob: Used Computer-2. Bluetooth address: 00:15:83:43:5F:40

3.1.3 Ubertooth

The Ubertooth's capabilities are described in 2, but from a hardware standpoint it is a 2.4 GHz transceiver. It is not recognized as a Bluetooth device but is able to observe and decode Bluetooth traffic. It has the hardware requirement to send Bluetooth packets, but no implementation for sending real packets is developed as of yet. There is some test functionality that lets you send a "Bluetooth-like" packet, which mimics a real packet, but due to the timing constraints on the encoding and decoding of packets the fake packets will not be recognized by a Bluetooth device.

3.1.4 Antennas

For the range testing of the Ubertooth two antennas was uses. The first one, denoted Antenna-1, was the standard antenna that came with the Ubertooth. This is an 8 cm antenna found on wireless network cards and other 2.4 GHz equipment. This was the antenna used for all of the testing except the second part of the range testing.
Antenna-2 was a larger antenna designed to boost the range of a 2.4 GHz signal with a 5 dBi gain. The antenna was used to demonstrate the possibilities of added range when monitoring traffic.

3.2 Software

The Ubertooth software is split into two main parts. One part for Bluetooth specific operations like FEC decoding and hopping sequence generation and one for Ubertooth management and bit stream handling. For the sake of simplicity the code is from now on referred to as host code and firmware, where the host code will be equivalent to a Bluetooth host, and the firmware the a Bluetooth controller.

3.2.1 Host code

The host code contains all of the tools running locally on the host machine and is utilizing functionality on the Ubertooth via USB. The host code is where the packets revived by the Ubertooth are processed. This includes address recovery, packet decoding and extracting information from the header.

Tools

The Ubertooth has a number of tools for monitoring Bluetooth. The complete list as of writing is:

- ubertooth-lap: LAP recovery

- ubertooth-uap: UAP recovery
- ubertooth-hop: Hop sequence recovery for a piconet
- ubertooth-specan: Signal strength readings for the 2.4GHz spectrum
- ubertooth-specan-ui: Graphical representation of the signal strength readings
- ubertooth-dump: A bit stream dump
- ubertooth-utl: Utility tool
- ubertooth-dfu: Firmware flashing tool
- ubertooth-ble: Bluetooth Low Energy monitoring

This list is subject to change due to the constant development of the Ubertooth, and has changes several times during the workings of this thesis.

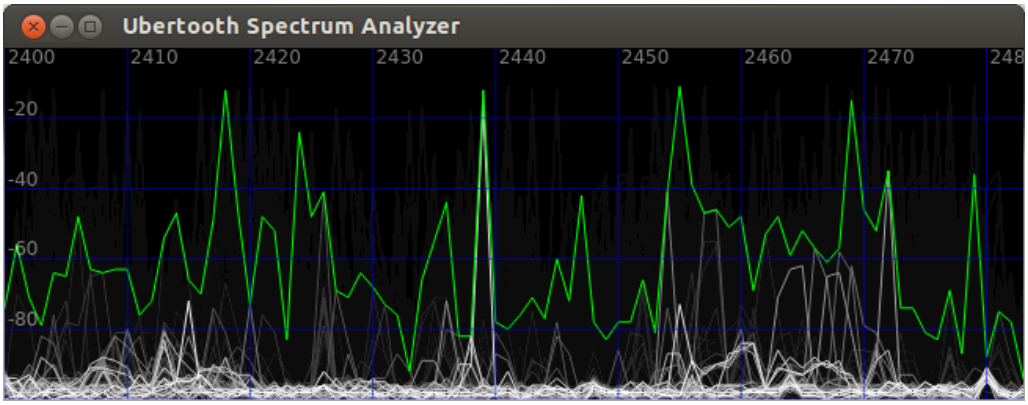

Figure 3.1: Ubertooth-specan-ui

Figure 3.1 shows ubertooth-specan-ui. The spikes represent active Bluetooth traffic, much of the other traffic that can be seen is two active wireless-LANs.

Hopping sequence

The hop sequence is discovered by using the observed traffic to determine the correct clock value. A large number of possible clock values are generated and the values are used to try to unwhiten a packet. Values are removed if they don't produce the right UAP and get the CRC-check correct. If the entire list is tried with no positive result the list is reset and a new packet is used to try to get the clock value. Once the correct clock is found the hopping sequence can be calculated using the master address and the clock value. The Ubertooth

then uses the hopping sequence to hop along and only observe traffic from the targeted piconet. When a packet is revived it is attempted decoded, meaning that the packet is unwhiten and the FEC encoding is reversed. The packet type is determined and information from the header and the payload is printed. The Ubertooth is however not capable of decoding every packet it observes.

Kismet plugin

The kismet tool is a way of monitoring the traffic and confirming the LAP and UAP of active Bluetooth devices. A lot of the observed traffic will be from a single address (9E:8B:33). This address is a broadcast address used by all Bluetooth devices. There will also be a steady stream of new addresses discovered. These are false positives identified by the Ubertooth and can be ignored. This means that any monitoring is reliant on receiving a number of packages before any certainty of the addresses of an active piconet can be confirmed. There will also be two package counts related to any adapter, one with only LAP, and one with LAP and UAP. This means that four of the lines displayed by Kismet will represent one active Bluetooth connection.
Kismet also logs the traffic as .ppt files that can be read by Wireshark for further analysis. These log files structure the information captured, separating the payload and the header making an analysis easier. This could be useful for analyzing a large number of captured packets. It also makes it possible to sort out traffic from a single source, even though Kismet captures data from multiple sources.

Packet information

The Ubertooth displays a fair amount of information about the packet it receives. Once the correct clock value is determined and it begins to decode it gives information about the header, but not the content of the payload of a data packet. The header includes the link layer address, link layer id, flow bit and payload length.
The payload of signaling packets are however sent in clear text and useful information can therefore read from these. Knowing the type and structure of specific packets makes is possible to extract useful information about the piconets Bluetooth channels such as channel ID's. The task of recognizing different packets beyond the basic packet types is however not at thing the Ubertooth does yet.

Figure 3.2 shows the Ubertooth acquire the master clock and starting to hop along with the piconet and decoding packets.

```
Calculating complete hopping sequence.
Hopping sequence calculated.
26519 initial CLK1-27 candidates
26519 CLK1-27 candidates remaining (channel=39)
688 CLK1-27 candidates remaining (channel=39)

GOT PACKET on channel 39, LAP = 436025 at time stamp 75858826, clkn 12137
15 CLK1-27 candidates remaining (channel=39)

GOT PACKET on channel 39, LAP = 436025 at time stamp 76390120, clkn 12222

Acquired CLK1-27 = 0x654e20c
got CLK1-27
clock offset = 212437300.
systime=1355165116 ch=70 LAP=436025 err=0 clk1=0x00036fc s=-54 n=-54 snr=0
Packet decoded with clock 0x00036fe (rv=1)
  Type: NULL
Packet decoded with clock 0x00036c1 (rv=1)
  Type: POLL
  Type: POLL
systime=1355165117 ch=57 LAP=436025 err=0 clk1=0x0003ac6 s=-68 n=-57 snr=-11
Failed to decode packet
systime=1355165117 ch=14 LAP=436025 err=1 clk1=0x0003ad5 s=-63 n=-59 snr=-4
Packet decoded with clock 0x0003ad7 (rv=1)
  Type: DH5/3-DH5
  LT_ADDR: 1
  LLID: 3
  flow: 0
  payload length: 574
  Data:  c7 a9 3e c7 e1 55 e1 a2 cb b5 fb b1 7f c8 3d ef d0 ff fa 5e 80 da 54 8f
0 40 21 8c 0a 00 42 20 00 80 24 30 81 88 00 0c 00 c0 09 b0 00 40 a0 03 28 00 13
 db ff 34 0f f3 82 21 d5 80 e8 27 8e 97 fb a6 ea ee 7c 99 84 7a c2 93 69 43 f7 1
b9 dd a4 23 1c 72 e6 58 52 bf 86 b3 30 9d 16 9f df 6d d1 12 0d 3a 70 2f aa 5c c0
2 bf bc b0 3a 5b 40 40 1a 58 41 c5 45 ec 8a ed c7 24 46 92 d5 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 00 00 00 00 00 00 0
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
0 00 00 00 00 00 00 00 01 10 11 00 70 eb 6c bc cb 50 e6 7d 9d 6e be 4a 85 82 ff
 8e f7 96 bb cb 79 62 60 5f 90 c6 e5 94 57 e2 23 96 3f 10 a0 d1 c9 77 da d5 e0 7
30 66 16 58 88 0c 9f cb 21 fe dc bf 02 0e f0 b0 b1 73 84 b0 29 73 0e d3 5d 46 b7
b 0e 68 65 ce 3d 99 88 cd 7e da 33 93 c6 ff ec 3b d4 95 ea 3a 7e fc 06 ff 4f 78
 b3 e4 2f ea 72 5e 01 f2 57 b0 e0 36 cc fd 3f 5e 28 37 eb f5 0a a6 e4 65 fc 51 8
Failed to decode packet
systime=1355165117 ch=25 LAP=436025 err=0 clk1=0x0003cb4 s=-68 n=-61 snr=-7
```

Figure 3.2: Ubertooth-hop

Packet encoding

The Ubertooth contains code to reverse the whitening, HEC, FEC and calculate the CRC. The tools only use this to decode, but it possible to reuse some of it to craft a packet. The unwhitening code can be used to whiten as it only XOR's data. There are also a function that FEC encodes 10 bits of data at a time.

3.2.2 Firmware

The firmware is code that is loaded onto the Ubertooth-chip and acts as a link between the hos code running on the pc and the hardware on the chip. The core

functionality can be broken down to receiving Bluetooth packages and sending "Bluetooth-like" packages. The code as it is from the developer site contains only bit stream processing. There is a small amount of test code for using the Ubertooth for doing range tests, but this requires two Ubertooth and will only mimic Bluetooth traffic.

3.2.3 Software Limitations

The main goal for this thesis was originally to develop code to prove a attack on Bluetooth is possible. The main reason for that this was not accomplished is development state that Ubertooth is in.
At the startup of this thesis the Ubertooth did not provide functions vital for making an attack work. The main function being the hopping sequence retrieval. This was added to the development code, but it was not entirely stable. A bug in the code caused the Ubertooth to crash as the master clock was retrieved. This particular bug proved to be hard to reproduce by other than the author. The developers of the Ubertooth, mainly by Dominic Spill, tried to track down and fix this bug, but upon the completion of the thesis the bug still persists. As far as the author knows, no one else is experiencing this particular bug. To rule out any local factors multiple hosts PC's was tried, and a second Ubertooth was tested. None of the testing produced any local variable that could be cause of the bug. A workaround was eventually found by combining old and new code.
The time spent digging around after a possible cause gave valuable information on how the Ubertooth and the code work, but left little time to develop.
The core code as it is today has not a lot of documentation. Most functions are explained by the name and the comments, but the interaction between them is not explained in any way. The architecture is similar to a Bluetooth device, and the naming convention for the functions relates to the Bluetooth specification, but this still require comprehensive background knowledge to understand.
The main function of the Ubertooth today is monitoring traffic, this means that all of the code i written to decode an extract information. All of the packet handling is done on the host side of the code, while injection requires that some of this is handled on the firmware side. Crafting a packet to inject requires reversing most of the functions used to decode, some can reused as it is but not all.

3.3 Capability Testing

When testing the Ubertooth capabilities two setups were used. The basic setup used the two Bluetooth adapters with the computers. The second used Computer-1 and the mobile phone to generate traffic at a distance from the Ubertooth. In

both cases a large text file was used to generate a continuous stream of traffic. The native OS Bluetooth software used numeric comparison when setting the connection. The PIN number used was generated by the application layer and required only the author to compare the two numbers presented on the screens. The native OS software also gives the complete Bluetooth address for any discoverable device in range; this was used to confirm the addresses of the USB-adapters before any testing started. Between each of the test the USB-adapters was un-paired and the Ubertooth was reset to ensure no interference between the tests.

3.3.1 Basic Operations

When testing general capabilities the basic setup with the two computers was used. Both the computers where in close proximity of each other to ensure minimal interference.
The kismet plugin was used as an initial address discovery. This confirmed that the Ubertooth can detect Bluetooth devices in non-discoverable mode. This also demonstrated the number of false positives the Ubertooth generates, as the kismet plugin showed an ever increasing list of detected addresses. These addresses were either close to the one used by the adapters, or completely wrong. The common thing about all of these addresses was the low number of packets associated with them. This could be either real traffic unintentional intercepted or just false positives.

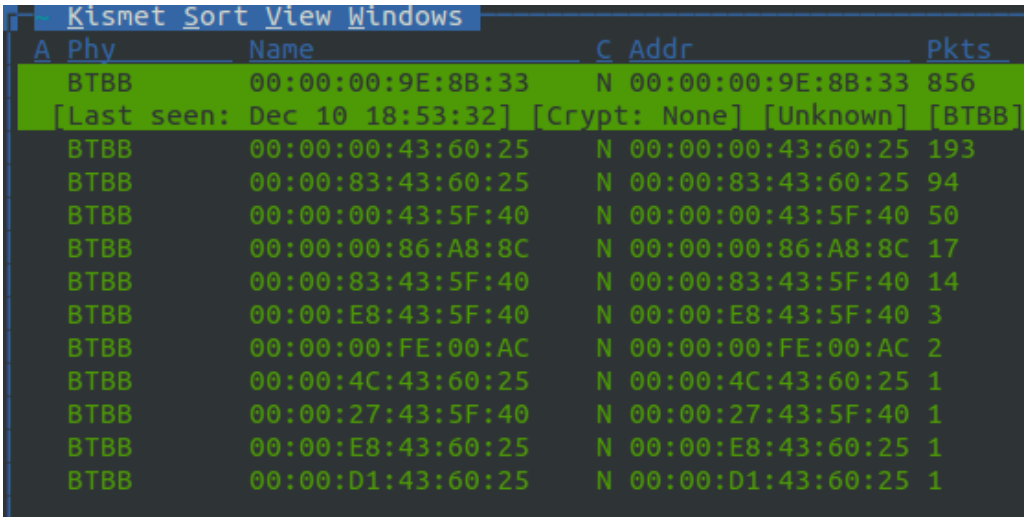

Figure 3.3: Kismet discovering devices

Figure 3.3 show Kismet discovering devices. The top line is the general discovery address. Note that there was only one discoverable device present

Attempt number	Number of packages
1	180
2	80
3	28
4	10
5	8
6	372
7	8
8	4
9	164
10	25

Table 3.1: Number of packet before hopping sequence was found

when this snapshot was taken. The second and third line represents Alice, which is discoverable. While the fourth and sixth is Bob who are set to be non-discoverable.

Obtaining hopping sequence

Ubertooth-hop with the LAP and UAP as input, was used to obtain the hopping sequence, and note the amount of information available with minimum setup. Getting hold of the hopping sequence means that you have to have traffic to test the possible master clock value against. An idle piconet will generate some traffic without the user interacting by sending link level packets[1] packets. This can be used to get the clock value as any other traffic, but the amount of packet is less than a file transfer will produce. The time the Ubertooth used to obtain the hopping sequence varied greatly. Table 3.1 is the results from 10 consecutive attempts to obtain the sequence. As can be seen clearly the number of packets required varies with several hundred. The average number of packets required in this trail was 87,9.

3.3.2 Range Testing

The range testing used Computer-1 and a mobile phone. The phone was positioned 15 cm away from the USB-adapter, and the two devices were moved at 5 meter increments away from the Ubertooth. The test was performed on campus at NTNU and would be equivalent to a real world office scenario with multiple wireless-LANs and walls present. To generate traffic a 1 MB pdf was sent from the computer to the phone. The test was performed when the Ubertooth already had obtained the hopping sequence. The measuring was done five times at each

[1] These are NULL and POLL packets used to maintain the link.

Meters	Average observed	Max observed	Min observed
0	33	56	20
5	22.8	38	13
10	15.8	21	6
15	21	32	7
20	11.2	18	6
25	8	20	4
30	5	6	2
35	3.2	9	0
40	0	0	0
45	0	0	0
50	0	0	0
55	0	0	0

Table 3.2: Measurements from Antenna-1

location, and the average of these was calculated. The number of packets intercepted by the Ubertooth at each distance was recorded. These are packets that belong to the targeted piconet, but what kind of packets and information available in them is not recorded for this test. The Ubertooth was used with the standard antenna which it comes with, and a larger 2.4 GHz antenna. The closest measurement was done at less than 1 meter from the Ubertooth and the furthest was 55 meters for both antennas.

Antenna-1

Antenna-1 is a standard 2.4 GHz antenna, 8 cm in length, commonly used with a wireless network card or on a wireless switch. The furthest distance which gave reading with Antenna-1 was 35 meters. The tests at greater lengths gave no reading at all. Table 3.2 sums up the measurements for Antenna-1, for the complete set see Table A.1 in Apendix A.1. Note the difference between maximum and minimum number of packets in the table. The lowest number with the closes range is the same as the maximum at 25 meters. This indicates that the measurements are effected by uncontrolled factors in a real life environment.
The blue line in Figure 3.4 shows the average number of packets for Antenna-1. There is a slight dip at 10 meters and a peek at 15 meters. This is most likely local radio conditions interfering and making the measurements uneven.

Antenna-2

Antenna-2 is a 2.4 GHz, 5 dBi antenna designed to extend range of wireless equipment. Table 3.3 sums up the measurements for Antenna-2, for the complete set see Table A.2 in Apendix A.1. The furthest distance packets was ob-

Meters	Average observed	Max observed	Min observed
0	77.6	94	54
5	46.8	79	25
10	31.6	64	21
15	35.2	58	23
20	25	40	17
25	9.6	11	8
30	4	16	0
35	19.2	27	7
40	5	12	0
45	10.2	27	0
50	0	0	0
55	0	0	0

Table 3.3: Measurements from Antenna-2

served was 45 meters. Like in the measurements from Antenna-1 there are variations in the number of packets observed at each distance. The red line in Figure 3.4 shows the average values for Antenna-2. This also shows the same dip at 10 and peek at 15 meters, which confirms the local conditions interfering in the same way as with Antenna-1. It also shows peeks at 35 and 45 meters, this is again a local variation, and are interesting because of the large number of packets that where intercepted at that distance.

The antennas compared

The graph in Figure 3.4 shows the results from the range test for both antennas. This clearly show Antenna-2 being a lot better at close range, over doubling the number of packets observed in average. Antenna-2 also captured the same amount of packets at 35 meters as Antenna-1 did at 15 meters. The distance Antenna-2 captured data at was only 10 meter further away that with Antenna-1, but the number of packets captured at the end of the range was significant. Being able to capture an average of 20 packets at 35 meters is almost the same as Antenna-1 had at 5 meters. Antenna-2 it therefore not only able to extend the range, but is also more efficient of capturing data.

Traffic monitoring

While following a piconet all of the intercepted packets are attempted decoded. Since there is no support for EDR traffic only parts of the packet stream are visible. A decoded packet will display header fields and the payload if there is one. The payload displayed by the Ubertooth is in hexadecimal form, but is still encrypted. A great deal of the packets observed will be NULL and POLL packets,

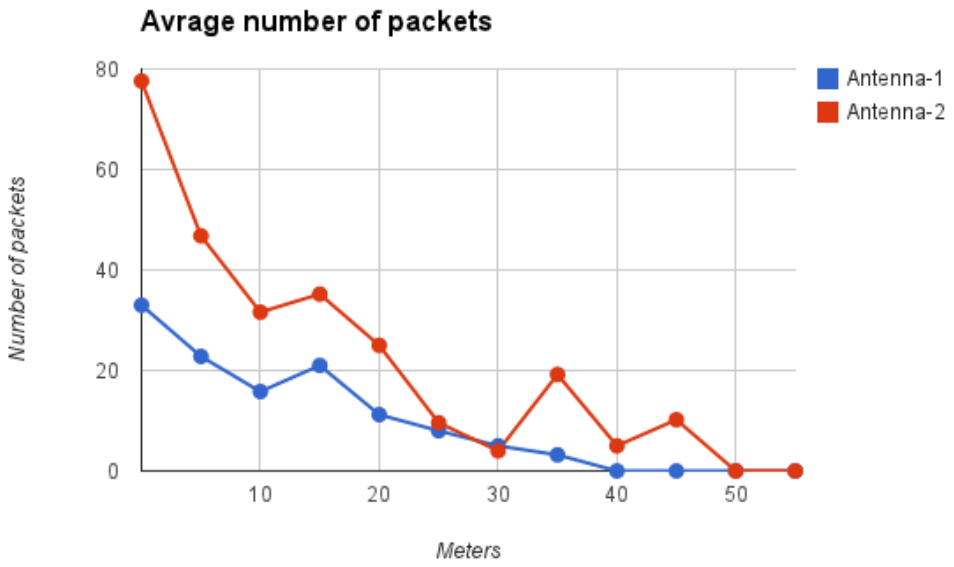

Figure 3.4: Average number of packets observed

which contains no payload. These are LMP level packets that is used to maintain the link.
The Ubertooth reported every time it received a packet belonging to the targeted piconet, but will only display information about the packet if it can decode it.

Decoding here means that the payload format is recognized. The packet is attempted un-FECed and then unwhiten. If the packet type has a payload header this is attempted processed first, given a success the rest of the payload is attempted processed. If any of these steps fails the operation breaks and no information about the payload or header is given. A EDR packet would not be decoded.

Table 3.4 shows the summary of 10 tests done with Antenna-2 at less than 1 meter, to see the average number of packets that are decoded. The same file transfer used to do the range test was also used here. The numbers of packets that are observed vary a great deal more in this trail than when doing the range test, but the total average is about the same. See line 1 in Table 3.3 for a comparison. The test shows that the average number of decoded packet is 86%, and the number of failed decodes is 14%.

Test number	Packets decoded	Packets not decoded	Total packets
1	75	13	88
2	62	15	77
3	130	3	133
4	29	2	31
5	27	6	33
6	20	10	30
7	102	14	116
8	82	10	92
9	74	23	97
10	76	15	91
Average	67.7	11.1	78.8

Table 3.4: Number of packets decoded

3.3.3 Processing Captured Information

To make any use of the information captured the packets has to be recognized. The Ubertooth gives information about the basic packet type, but nothing about the content of the payload.

Recognizing known packets

In the event that a packet is decoded and the payload is displayed, it still has to be manually decoded. The software only separates out the fields in the payload header. The payload header holds the LLID field that indicates which of the logical channels that are used which can be useful for identifying the payload.
A DM1 with a LLID equal to three would indicate ACL-C packet. This will always contain LMP PDU.
A DM or DH packet with a LLID equal to two or one would indicate a ACL-U packet. This could contain any kind of information, and could be encrypted. If a LLID is set to one, the PDU is fragmented and indicates that the current packet is a continuation and contains only parts of a PDU. With the LLID set to two an entire PDU may be contained in the payload.

Decoding by hand

As the software does nothing more than recognize the types of packets, one would have to decode the packet by hand. Any ACL-U packet may be encrypted and may break the PDUs into several packets this is almost impossible to extract useful information.
A ACL-C packet is possible to recognize and decode by hand. The LMP PDUs is well defined, but the number of that kind of PDUs is large and only contains

link management information.

Chapter 4

Attacking with the Ubertooth

In this chapter attacks on Bluetooth with the use of the Ubertooth is presented. The first section is a look at passive attacks. The second section proposes active attacks that can be performed by the Ubertooth.

4.1 Passive

Preforming a passive attack on Bluetooth is a major challenge. The way the traffic is obscured form any observer makes the capturing of data a task in itself. The Ubertooth makes this a lot easier since it can follow a piconet and decodes parts of the traffic it captures.

4.1.1 Traffic Monitoring

The main problem with monitoring a packet stream is the time it takes the Ubertooth to obtain the hopping sequence. The time this takes and the number of packet required varies greatly. This is caused by two factors, one being that the traffic can be subject to interference and there is no way of being sure where in the packet stream you start to observe. The second is that a large number of clock values have to be tested. Basically one makes an educated guess what the clock value of the master is, and tries values based on the guess.
As seen in Chapter 3, the Ubertooth may require a number of packets, or a connection that persists over several minutes to begin to monitor the transmission. When the hopping sequence is discovered the number of packets actually decoded is varying. The Ubertooth does not support EDR, meaning that any connection using EDR would just be subject to partially monitoring. This means that even with the Ubertooth following a piconet the amount of data you can extract is limited. The payload itself is also encrypted with a stream cipher, one

that has no practical attacks against it, except the zero-knowledge attack presented in Chapter 2.

4.1.2 Location Tracking

Location tracking with the Ubertooth means targeting traffic belonging to one or several devices. The Kismet plugin could be used to do general observing in an area.
It is possible to target a single device even if it is part of various piconets. The input parameters used by the Ubertooth is the address of the targeted device, no matter if it's the master or not. The challenge here is to obtain the correct address and that the targeted device has to generate traffic. As seen in Chapter 3, the number of packet captured decreases by the distance you are from the targeted device, this sets constrains on how close you have to be to a target.

4.1.3 Stream Cipher

The stream cipher used by Bluetooth is still considers secure, and there is therefore no direct use of the Ubertooth in decoding the payload of normal packets. It may be used to record data to use in a future attack, and would be well suited to do just that.

4.2 Active

Using the Ubertooth to do an active attack is possible, but there is no implementation for a injection attack currently available. The following text describes attacks which is possible to perform with the Ubertooth, given the right implementation and under the right circumstances. The attacks are very similar in nature and only differ in the layer they are targeted against. The first part of the text underneath describes a Denial of Service (DOS) attack, based on injecting correct packets. This explains how it can be done, what values that are needed, and how it can be implemented. The last part explains what attacks can be done if the DOS attack succeeds, and the implications of this.

4.2.1 Proposed Attack

The attack proposed here is a denial of service attack against a specific piconet. Using a disconnect packet to shut down the Bluetooth communication. A dos attack is very basic but it can be a starting point for other attacks. Given that one can prove that Bluetooth injection is possible, one can craft other types of packets and preform more elaborate attacks. The proposed attack in its basic form will be performed in the following manner:

- Obtain the LAP and UAP of one of the devices in the piconet. By using the kismet plugin or ubertooth-uap/-lap tools.
- Use the ubertooth-hop code to discover the hopping sequence and hop along with the piconet.
- Edit the fields in a observed packet, or use a premade packet.
- Encode the packet.
- Send the packet to the targeted piconet.

If the attack is successful the observed data stream would disappear. This would prove that it is possible to forge Bluetooth packages and get them accepted. Note that this is a packet sent with Basic Rate as this is the only rate supported by the Ubertooth. This attack does not compromise the data sent over link in any way. A extended attack could potentially do just this, exploiting that the piconet has to establish a new link. Making it possible to observe the link setup procedure and extract information used to extend the attack.

The attack proposed here can be performed with two types of packet, depending on what layer it's possible to inject packets in. A LMP packet would target the link layer and disconnect the physical link. A L2CAP packet would tear down the channel but maintain the link. This attack could be a more discrete way of launching a MITM attack. A lost connection would from a user's perspective look like a bug, and may not alert the user to the presence of an attacker.

4.2.2 Initial Attack Phase

In both cases addresses of one of the devices in a piconet is needed to launch its initial phase. Knowing that an address belongs to the right device would be a task in itself and is outside the scope of this thesis. Observing an address that is in use is easy with the Ubertooth. Even with the Bluetooth device in non-discoverable mode will be visible, as long as it's transmitting some type of data. The addresses gives the basis for hop-sequence discovery, but is hinged on the amount of traffic observed. It is possible to obtain the sequence from a small amount of traffic, but that may require a larger amount of time.

4.2.3 Obtaining The Header

The header to both attack packets would be identical, and a valid header could be used in both cases. One possibility would be to observe a packet and reuse parts of it. Any decoded packet would do as the direction of the packet doesn't

Field	Size	Value
Access code	72 bit	Reused or constructed
Header	18 bit	Reused or constructed
Payload Header	8 bit	0xE02 (LLID=3, FLOW=1, Length=2)
Transaction id	1 bit	0x1
OpCode	7 bit	0x07
Error code	8 bit	0x1F
CRC	16 bit	Must be calculated

Table 4.1: Complete DM1 detach request packet

affect the header. It's possible to reuse most of the header from a observed packet because most of the header values would be static for a given piconet. It is therefore also possible to craft one by hand if the target address is known. The only header value that is dependent is on the rest of the packet is the type field. The header must therefore be modified to match the type of packet that is used. The Ubertooth code currently has code that makes it possible to craft an almost correct header, given that the address of the master is known. The only value that can't be calculated with the current code is the HEC.

4.2.4 Constructing a Detach Request

A packet with the right header would be accepted by a Bluetooth device, but getting the a disconnect request to be accepted requires a bit more information. The payload header has to be correct for the PDU that is sent. The LLID field has to be set to 3 to indicate a ACL-C packet, the flow bit is set to one but will be ignored. The length of the PDU is set to 2 bytes.
A LMP detach PDU requires only three fields and they are all well-defined. First is the transaction ID, this is one bit and could be set to both 1 and 0, as the direction of the packet isn't a concern. A detach request would be accepted by both master and slaves. The second field is the OpCode, and has a value of 0x07 for the detach request. The last field is an error code field. From a attackers point of view this may be set to any of the predefined error codes. The code 0x1F which means "unspecified" seems to be a good enough choice. An LMP_detach_req is always sent as a DM1 packet, and would require a header that has the right type field. Table 4.1 sums up the fields in a DM1 detach request packet, their values and where the values come from.

4.2.5 Construction a Disconnect Request

To generating a disconnect request that will be accepted means making a L2CAP PDU. Most of the packet can be prepared except the CIDs contained in the pay-

Field	Size	Value
Access code	72 bit	Reused or constructed
Header	18 bit	Reused or constructed
Payload Header	8 bit	0xAA (LLID=2, FLOW=1, Lengt=10)
Channel id	16 bit	0x0001
Code	8 bit	0x06
Identifier	8 bit	0xFF
Length (of data fields)	16 bit	0x0020
Data (DCID, SCID)	32 bit	Must be observed and reused
CRC	16 bit	Must be calculated

Table 4.2: Complete DM1 disconnect request packet

load. For a given Bluetooth channel there are assigned two ID's to define the two endpoints, denoted DCID and SCID. As these are values that are dynamically assigned by the master upon channel establishment they must be observed and reused. The main problem is therefore to observe and recognize a L2CAP PDU. The channel IDs are used in all L2CAP PDUs, and are contained in the L2CAP header in the second and third octet. To get both endpoints in a piconet one would have to be able to recognize the correct PDU and extract the correct bits. Two recurring values in the same position would signify the presence of the CIDs. Note here that this only holds for a piconet with one slave, as each slave would have its own CID. One must also take consideration of reserved CIDs like the signaling channel.
The CIDs are also used in three signaling packets, the disconnect request, the connection response and configuration response. The two last are both packets sent at the channel setup, so observing these could be a challenge. Observing a disconnect request would not be useful as the channel then would disappeared. Therefore to get the CIDs from a signaling packet one has to find a packet with payload containing channel id=0x001, and with the right response codes.
After the correct CIDs is obtained and inserted into the payload the CRC has to be computed.
To send a L2CAP signaling packet we need a suitable packet type, it must be basic rate and support the right length of the payload. Since the length of the PDU that is to be sent is only 10 bytes a DM1 packet can be used. In the payload header the LLID has to be set to 2, as this is a ACL-U packet, and the flow bit are set to 1. Table 4.2 list up all of the fields in a DM1 disconnect packet, its value and where the value comes from.

4.2.6 Encoding The Packet

Crafting a packet that would be accepted as a Bluetooth packet also requires the right encoding. It is not possible to just replay a revived packet because of the whitening. The whitening is dependent on the clock and a replayed packet would be whitened with the wrong clock value.
The main task for constructing a packet would therefore be to whiten and then FEC encode it. Since the clock is required for whitening this must be done with an active connection.
The both header and the payload in a DM1 and DM3 packet is whiten and 2/3 FEC encoded before transmission.

4.2.7 Implementing The Attack

When implementing this attack in code timing is a concern. Most of the packet can be prepared without consideration of the time spent, but not the encoding. The header values and the payload, along with the CRC calculation should be done on the host side, as none of these values are dependent on timing. The whitening is done before the FEC encoding and the packet must therefore be sent reasonably quick after the encoding is done. This must therefore be done on the firmware side.
For the LMP packet everything can be prepared before the attack begins. The only thing that must be done live is the encoding.
For the L2CAP packet the CIDs has to be observed and reused. This requires a method that can recognize a L2CAP signaling packet and extract the values. The alternative is to guess the CIDs, or try to brute force them.

The optimal solution would therefore be to observe a L2CAP PDU with the CIDs in the payload. Extract the values and insert them into the prepared packet. Calculate the CRC. Send the packet to the firmware, whiten and FEC encode the entire packet, then send it.

4.2.8 Additional Attacks

Being able to successfully inject a packet will open the possibility for other attacks. The detach request is a LMP PDU, and there are other LMP PDUs that can form the basis for attacks. The most interesting may be the PDUs that handle encryption. Both the ”pause encryption” and ”stop encryption” are interesting, but both poses its own challenges.
The L2CAP PDUs used for signaling is also interesting but is limited in the number of valid options availible.

Link Manager Protocol PDUs

The LMP_stop_encryption would be the most interesting to get accepted, as it would make an attacker able to listen into any traffic sent. The main problem here is that it can only be sent from master to slave. A slave has no opportunity to make a master switch off encryption. This makes it impossible to use as an attack, besides a type of denial of service attack disabling encryption on one side.

The LMP_pause_encryption may be an easier target, as it can be sent from both master and slave.

- Option 1: A ”pause encryption” received by the master. The master will respond with a ”stop encryption” response to the slave. The encryption is thus paused on both sides.
- Option 2: A ”pause encryption” is received by a slave. The slave will send a ”pause encryption” in to the master and Option 1 occurs.

In effect both options halt the encryption with one injected packet, but there is still requirements to that packet. First off, there is a transaction ID required in the packet. The transaction ID is a one bit value and signifies the direction of the packet. If the bit is set to 1 it would look like a packet from the slave to the master, if set to 0 it would appear like the opposite. The effect would be the same; the encryption would be disabled in both cases. The main problem with both the master and slave in the ”pause encryption” state it that it does not allow user data being transmitted, only logical command messages is allowed in this state. The all of the ACL-U traffic is paused when the encryption is paused and can only be resumed when the encryption is turned back on.

The LMP_comb_key is the third interesting PDU, which changes the current link key. If the current link key is a unit key, pairing procedure will be done again, producing a new link key. This PDU can be sent from both master and slave and only contains a random number. By itself it does no harm, but it can still be exploited by a malicious observer.

L2CAP PDUs

The signaling sent with L2CAP PDUs is mostly interesting because of the disconnect request. The other signaling commands that is available only controls the flow of the traffic and does not interfere with user data directly. An accepted signaling message will however prove that it is possible to interfere with the traffic on a higher level. In the advent of a practical attack on the encryption this is the first step against interfering with user level data, as the L2CAP link does

not only carry signaling commands but user data as well. Being able to inject falsified information into user data would create a whole new attack vector.

Chapter 5

Analysis

In this section the analysis of the Ubertooth as a tool for improved attacks on Bluetooth is presented.

5.1 Why Use The Ubertooth

The Ubertooth is a monitoring device that differs from other available sniffers. It's for one not reliant on prior knowledge about the devices that are targeted. It is also entirely open source and the hardware is available at around 100$. Compared to other methods it has distinct advantages.

5.1.1 Other Open Source

The attacks presented in Chapter 4 is for the most part performed by Bluetooth adapters that you can change the address of, and thereby trick legit devices. The problem is therefor that you have to get the address somehow, reprogram the firmware of your device. Then you need to get the targeted device to pair with your own device before you get anything useful out of it.
Getting the address of a device, using a standard unmodified adapted, is possible. The problem is that the target has to be discoverable. A non-discoverable device will not be visible to a standard adapter.
If you can get the address there is still the problem of pairing it with your own device. This means that you are reliant of a user interaction, and that the user is fooled by your pairing.

5.1.2 High End

There are high end professional devices capable of doing Bluetooth monitoring. The can give you the complete traffic stream, and even decode encrypted traffic. Using this in an attack scenario seams perfect, except it requires information that

is unrealistic to obtain as an attacker. As this is a developer tool it's not designed around detecting traffic and devices. To be able to observe a conversation from the very beginning the native clock and full address of the devices is needed. To be able to decode traffic the encryption key is required. As an attacker you will most likely not be able to get this, not in a realistic scenario anyway.

5.1.3 Advantage From Other Sniffers

The Ubertooth give the advantage of being able to detect any active Bluetooth device in range, non-discoverable or otherwise, and get the address. It also gives you the possibility to lock on to a device and follow its traffic stream, using only the address as input.
Ubertooth is also open source, meaning that anyone can access and modify the code. Even the schematics for the hardware is available, and the hardware cost will only set you back about 100$. Buying one premade is not that much more expensive, and are available from several web shops.
The cost, its functionality and the possibility to modify the code as you see fit separates it from other available sniffer.

5.2 Ubertooth Capabilities and Limitation

The Ubertooth is a good tool for monitoring Bluetooth traffic, but there are limitations on the capturing of the data. The main problem is the instability in the amount of traffic observed. In chapter 3 both the range test and the decode test shows grate variance in the number of packet observed. This is despite that the test was done with little delay between them and with no change to the setup.

5.2.1 Range

The range of the Ubertooth is equivalent to a Class 1 Bluetooth device, and has a maximum range of 100 meters. The practical distance decreases with any disturbance and noise created by the local environment. As shown by the results in Chapter 3 the local conditions have a lot to say when it comes to intercepting traffic. Another environment may give an attacker both better and worse results. The range test was done against a known target and with the Ubertooth already following the piconet, which does not mimic a realistic attack situation. Despite this the results obtained is still interesting as this was done as a test of range, and not information extraction. It also gives an indication to how much information that is possible to get at range.

Antenna-1

As seen in Chapter 3 the range of the Ubertooth even with a small antenna is significant. The test done showed that it it's possible to intercept traffic at a distance of 35 meters in a office environment. The amount of data captured at 35 meters was not large, but it could still provide information of a device. The loss of information increased quite rapidly with the distance, meaning that any attack focused on the information in the packet stream would be hindered. Any attack reliant on large amounts of data would require a closer range or a greater time span.

Antenna-2

The test results from the second antenna show that it is possible to intercept traffic at distances up to 45 meters. The average number of packets captured at 35 meters was 19.2, and had a maximum at 27 packets. This is more or less one fourth of the traffic captured at less than 1 meter, and gives an attacker considerable range even if it's at 75% loss.

Implications

The average loss of data increases a lot with distance, and one have to be lucky to observe an entire packet stream. One would not be able to capture entire files, and certainly not at greater distances. This does not however guarantee that the information captured is not critical. An attacker looking for information about the devices in an area, or looking for someone specific, would not be hindered by loss of data as long as it's possible to observe some traffic.
Being able to listen into traffic going on in a radius of about 45 meters away from you is a lot. There could be a great number of people with Bluetooth enabled devices in a sphere that big. Note that none of the measurements was done through floors or with a vertical aspect, only through walls in a horizontal fashion. Going through floors will probably put extra constrains on the practical distance, but that does not make the measurement any less interesting.
In the author's opinion, the range of the Ubertooth is impressive in a real world scenario. Despite the fact that its equivalent to a class 1 device but could not detect traffic at 50 meters, being able to listen in on a packet stream at half the maximum range in a real world test is quite the feat.

5.2.2 Tools

The tools provided with the Ubertooth when writing this has limited use. They provide basic traffic monitoring, but the lack of functionality that can send traffic limits the areas it can be used. There is no way of extracting payload information other than decoding it by hand.

Payload extraction

As seen in Chapter 3 the average number of successful decodes of packets at close range are 86%. This implies that an attacker is not able to observe a complete packet stream, even if the target is unrealistically close. Any contents in the payload may be obscured because of this, depending on the type of information transferred.
Because the encryption used on the payload still is considered secure the payloads captured has little to no value as of yet. The packages that is transmitted with clear text payloads is still interesting, but exposes little information about what goes on above the link layer. The information from captured packets is presented in a manner that only persons with intimate knowledge of the Bluetooth standard will be able to read it. Given the development state of the Ubertooth this is not unexpected, but it narrows the amount of people that can use the information provided. This limits the number of people that can use it maliciously, but it also limits people with good intentions.
The inability to decode EDR traffic is also a hindrance, as most devices from version 2.1 and up uses EDR. There are still uses for the Ubertooth in its current state with EDR devices, but there are major limitations to it.
It is possible to decode some of the packet by hand, as some of the PDUs only are sent with certain packet types. This will however only give low-level information that is of little use. As of now it is no possibility to process the information captured by any other means.

Address retrieval

The addresses are a vital part of monitoring Bluetooth traffic, and the Ubertooth retrieves addresses with great user friendliness via the Kismet plugin. All Bluetooth adapters will broadcast the entire address while in discoverable mode, but the option to get a hold of the address of a device in any mode is still a vital improvement. All attacks directed at any one piconet will be based on retrieving the address to at least one of the devices in question.
The LAP is retrieved within seconds using the ubertooth-lap tool. The UAP requires the LAP and takes more time before it is confirmed, but does not require a large amount of traffic. Both address parts can be obtained with the tools with the minimal traffic an idle piconet generates.
Kismet is therefore useful in a general observing role, and the tools will be more useful when targeting something specific or from a development point of view.

Hopping sequence

The retrieval of the hopping sequence is in the authors' opinion one of the Ubertooth most useful tools. It lays the basis for both information extraction and in-

jection. The ability to follow a piconet takes care of one of the difficulties when it comes to Bluetooth monitoring. Even if not all of the traffic can be decoded it greatly increases the amount one can observe with one antenna. The time it takes to retrieve the sequence is still not a reliable thing and requires a varying amount of traffic. There are not all scenarios where one can expect a steady stream of data, especially if the paring between two devices has persisted for some time. The time used to obtain the hopping sequence has been improved during the writing of this thesis, pointing to the fact that it is still being worked on.

5.3 Improved Passive Attack

The one passive attack mention in Chapter 2 is not an attack that can be improved with ease. The nature of the presentation makes specific improvements impossible to present, but by analyzing the cipher used and taking in count the additional information Ubertooth can provide, a general improvement can still be proposed.

5.3.1 Stream Cipher

The cipher used in Bluetooth is a stream cipher that uses the address to the master device, a link key and the current clock value as input values. The address of the master is easily obtained with the Ubertooth, though this is not information that would be considered secret. The link key is on the other hand a well-kept secret. The last variable, the clock value, is normally not something that you could obtain, unless you are monitoring the traffic with a Ubertooth or paired with the target. The clock value is central for the hopping sequence and for the unwhitening, meaning that every packet the Ubertooth decodes has the right corresponding clock value.
The zero-knowledge attack presented in Chapter 2 shows that it is possible to obtain the cipher key used for a specific data stream. Meaning that three of the variables in the E0 computation is can be obtained.
This may give the basis for the Link key. Given that it is possible to perform the stream cipher attack on real traffic, and not just constructed traffic with known statistical properties. This may be possible to do if you can observe the connection setup, which is possible if you can perform a type of DOS attack.
A Bluetooth device will ask for services of a connected device, this will happen after the connection is set up, and therefore when the encryption is active. A service request is sent in a standardized fashion, and may contain the right statistical properties the zero-knowledge attack needs.
The main advantages here are that it is possible to record traffic for a given pi-

conet, and the corresponding clock value for each packet. If it's possible to obtain the cipher key for a given packet, that packet can be used to find a possible link key. If the link key is obtained the rest of the recorded traffic can be deciphered.
The E0 function has yet to be broken and may still be considered safe, but that does not make it safe for future attacks. The crypto analysis done in [8] point to a possible exploitable weakness. The presence of tools like the Ubertooth that provide information that is considers difficult to obtain in a real world scenario may even present other viable attacks.

5.3.2 Location Tracking

The Ubertooth makes it possible to observe Bluetooth devices in non-discoverable mode that are sending traffic. Paired Bluetooth devices will regularly send packets, and will therefore small but steady stream of observable traffic. If the address to a device is known the Ubertooth can be used to track or confirm the presence of that device.
Location tracking uses no other information than the address, and can therefore be done with minimal effort, but poses no threat against the victim other that privacy. In [17] one of the first things mention to mitigate passive attacks is to turn off discoverability on a device. This may seem too be a sound advice, but this does not provide real security from passive listeners, and may provide a false sense of security. Non-discoverable does not equal non-observer-able when transmitting something though the air.
Using the kismet plugin can also make it possible to observe behavioral patterns captured over a large amount of time. The capturing of data from a day, or even a week could provide information on when people are arriving and leaving a area. The presence of a device associated with a time stamp could make it possible to track the habits of people or companies.
In a location tracking scenario, being able to detect devices at a range of 45 meters in a office is quite significant. An open area environment would boost the detectable range and could disclose even more information.

5.4 Improved Active Attacks

5.4.1 Denial of Service

Making a DOS attack on Bluetooth is not that hard to accomplish. All you need is a microwave oven, or anything that can produce a lot of interference with the 2.4 GHz specter. A crude and simple method like that will also interfere with any Wi-Fi networks in the general area. In other words easy to notice for anyone with a basic Wi-Fi monitoring tool, such as a smart phone with the right

application.
Bluetooth is built while knowing that it uses a noisy channel, and compensates for this with techniques like FEC and whitening. It utilizes channel hopping to avoid using only the ones with mass interference. A more elegant dos attack would be to exploit the protocols own means of terminating a connection. This means that one would have to inject a Bluetooth packet, which has been proven to be hard to do, but definitely doable. Two disconnect packets is presented in Chapter 4 and both presents the possibility of a DOS attack.

LMP

The LMP packet presented in 4 has almost all of the required values. There is still a matter of generating the HEC for the constructed header. In addition it has to be whitened and FEC encoded at the right time. This is the most practical of the attacks and would be possible to do with the right implementation.

L2CAP

The L2CAP packet is lacking the values that have to be obtained dynamically and therefore the hardest to fake. The CIDs has to be extracted while the connection still exist and the CIDs are valid. As explained in Chapter 4, this can be done by observing L2CAP PDUs. There is also a possibility of doing a brute force attack to obtain the CIDs. By constructing a packet for each possible CID combination and send it to see if it will be accepted. The combination is is a 32 bit value, and that that means trying 2^{32} packets. The DM1 packet described in 4.2 would span one time slot, with 1600 hops/s and one time slot per hop, makes the time required a little over 31 days to send all packets. In other words an impractical amount of time for a Bluetooth connection. You may be able to do an educated guess on where to begin, and increment in a smart fashion, but it is not a very viable approach. This packet would also have to be whitened and FEC encoded before transmission, but this would be the same as to the LMP packet.

Implementing

The code to implement this attack is almost present in the existing Ubertooth code. There are no function that calculates the HEC, but a reverse function exists. The whitening can be used as is, and there is a function for FEC encoding. The code has to be adapted it so it takes in the prepared packets, and this would have to be done on the firmware, not in the host code as it is today.
The major challenge here is the timing of it all, since this cannot be done with-

out the hopping sequence. The Ubertooth has to get and maintain the correct sequence, transition from a listening state to a transmitting state, encode the packet and sent it. There is a function present in the current code which turns the Ubertooth into a repeater for ”Bluetooth-like” packets sent from a second Ubertooth. This may be utilized to make the switch from listening to transmitting, but this has to be invoked from the host code, as a successful decode is necessary to be sure of the hopping sequence.

5.4.2 Disable Encryption

With a proven injection attack as a basis there is the possibility of doing an attack that disables the encryption. As the encryption used still is considered secure being able to disable it on a lower layer would pose as a possible attack angle. The encryption is controlled through LMP PDUs, so it is similar to a detach message. Though disabling encryption it is a tempting target it gives no direct use, as the master has to send the ”stop encryption” message. It’s not possible to instruct a master to stop the encryption. Instructing the slave to stop encrypting is possible, if it’s only the information sent from the slave that is interesting. It is not entirely clear what would happen if the master suddenly only receives unencrypted packets, and how long the channel would stay unencrypted or active. In any case this would at best give a small amount of clear text information, and would rely on luck for it to yield anything useful. This would also almost certainly alert the user.
Pausing the encryption is possible from both master and slave, and both can be faked. The main problem here is that this would also pause the ACL-U traffic. Having no clear text user level traffic limits the uses of this attack. From what the author can see there is no command that can be given in this state that makes this attack interesting. Access to clear text link level information poses close to no threat against a user. This renders the attack to nothing more than another denial of service attack.

5.4.3 Changing The Link Key

If one can successfully fake a LMP_comb_key PDU, an new link key will be made. This in itself poses no harm, but it allows an attacker to observe the entire pairing procedure, or launch a MITM attack. The advantages of requesting a new link key would be that the entire process would require no user interaction. Setting up a MITM attack with this would leave out the user entirely, and would make the attack invisible.

5.4.4 Man In The Middle

MITM attacks presented in Chapter 2 has one thing in common, you have to know the address of both the target devices, and you have to program your own devices to mimic them. By the use of the Ubertooth getting the address from any device, even hidden is simple. The main problem these kinds of attacks presents, and is often ignored, is how to get the address in the first place. There is still the problem of knowing you got the right address, but you will be one step closer.
A well timed dos attack, where the addresses is obtained beforehand, makes it possible to stop a connection before it gets going. Stopping the connection while a user has the attention focused on connecting or sending, will most likely result in a rapid retry. When one retry is all that is needed before the sending succeeds the user will most likely be none the wiser that someone is trying to interfere with the transmission.
In the authors experience during testing of the Ubertooth, sending a file to another Bluetooth device does not always work on the first try. This happened multiple times, and there was no indication that this was caused by someone external. This indicates that this is caused by Bluetooth is self, a bug that probably is very common. So having to resend will probably not raise any suspicion with a normal user.

5.5 Implication of A Successful Attack

Being able to inject packets into Bluetooth traffic would at first glance not seem to be of any significance. The packets proposed injected here does nothing more that deny a user the use of a device for a short period of time. There is always the risk of someone using the attack as a basis for a MITM attack, but that wouldn't change the security situation significantly from where it is today. What it does, is lay the basis for other injection attacks. The stream cipher is not broken, but there is no guarantee that it will stay that way. Being able to access Bluetooth data on both higher and lower levels makes it possible to alter and monitor anything the user is doing. It is also possible that this would give access to higher functions on the device that implements the controller. The wide range of devices Bluetooth is used in makes this a concern. With billions of devices on the market makes the attack surface large. A major security issues that is uncovered may force a change in the technology, but the change will take time. The development of new security measures is not done overnight, neither is changing out all of the devices effected. This of course depends on the type of issue uncovered, but the number of effected devices would in any case be large.

Chapter 6

Conclusion

This project had as a goal to make use of the Ubertooth to examine Bluetooth and its security properties. A lab was set up to examine the capabilities of the Ubertooth and see how it could be used in a practical attack. The Ubertooth proved to give information about active Bluetooth adapters, and was able to detect traffic and display parts of the stream. The information captured is however limited in use. The information is displayed in its raw format and only the type of packet and the logical channel is separated from the rest of the payload.

The Bluetooth specification was studied and a practical attack was proposed. The attack is possible to preform, but the lacking possibility to send packet into an ongoing conversation hinders it. This functionality is however something that is being worked on and may be available in the near future.

The encryption of the payload is also a hindrance in getting information sent between devices. The E0 encryption used is as of today not broken. It may seem that it inhabits some weakness that is possible to exploit, but the exact nature of this is not yet public. What is known is that weakness uses some time to begin to retrieve encryption keys, and in a time sensitive attack this may not be usable. Recording traffic and decoding it later is however a real possibility, when weakness is made public.

The Ubertooth at its current state is best at monitoring and giving general information about the activity in the Bluetooth channel. The fact that is it available to anyone at reasonable price is also a advantages over other sniffers. It gives Bluetooth monitoring a more versatile tool those adapters with spoofed addresses. It also makes a new entry into basic surveillance with the possibility to detect ”hidden” devices. It proves that the only way a Bluetooth device is undetectable is if its turned off.

6.1 Further Work

There is still a lot that can be, and will be done with the Ubertooth. Making injection possible is a goal of the current Ubertooth community. The ”only” thing that is stopping it from being done at the time of writing is FEC encoding and whitening done at the right time.

There is also a need for a better way of extracting information from the packets captured. Logic separating packets that can contain valuable information and possible to detect values is needed for more advanced attacks.

Packet crafting code is also vital for any injection to work on a more general basis. Most of the information needed to craft a viable packet can be extracted as it is now. Recombining it in a known packet format will be possible to do, at least for the lower level packets.

Documentation of the Ubertooth code is perhaps one of the things most missed during the work done in this project. There are comments explaining the functionality, but an overall description and the interaction between the code parts is non-existent.

Bibliography

[1] i. Bluetooth SIG, "Simple. secure. everywhere." Retrieved 14 November 2012, from http://www.bluetooth.com/Pages/Simple-Secure-Everywhere.aspx. [cited at p. 3]

[2] *Network Security:Current Status and Future Directions*, ch. 18. Wiley-IEEE Press, 2007. [cited at p. 5]

[3] B. SIG, "Bluetooth basics." Retrieved 05 Desember 2012, from http://www.bluetooth.com/Pages/Basics.aspx. [cited at p. 6]

[4] B. S. I. Group, "Specification of the bluetooth system, core version 2.1 + edr." Retrieved 12 November 2012, from http://www.bluetooth.org/spec/. [cited at p. 13, 14, 18, 19]

[5] J. Alfaiate and J. Fonseca, "Bluetooth security analysis for mobile phones," in *Information Systems and Technologies (CISTI), 2012 7th Iberian Conference on*, pp. 1 –6, june 2012. [cited at p. 19]

[6] K. Haataja and P. Toivanen, "Two practical man-in-the-middle attacks on bluetooth secure simple pairing and countermeasures," *Wireless Communications, IEEE Transactions on*, vol. 9, pp. 384 –392, january 2010. [cited at p. 19, 20]

[7] J. Barnickel, J. Wang, and U. Meyer, "Implementing an attack on bluetooth 2.1+ secure simple pairing in passkey entry mode," in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on*, pp. 17 –24, june 2012. [cited at p. 19]

[8] E. Filiol, "Zero-knowledge-like proof of cryptanalysis of bluetooth encryption." Cryptology ePrint Archive, Report 2006/303, 2006. http://eprint.iacr.org/. [cited at p. 21, 22, 54]

[9] A. W. Ophir Levy, "A uniform framework for cryptanalysis of the bluetooth e0 cipher." Cryptology ePrint Archive, Report 2005/107, 2005. http://eprint.iacr.org/. [cited at p. 22]

[10] J. Dunning, "Taming the blue beast: A survey of bluetooth based threats," *Security Privacy, IEEE*, vol. 8, pp. 20 –27, march-april 2010. [cited at p. 22]

[11] M. Ossmann, "Project ubertooth." Retrieved 18 November 2012, from http://ubertooth.sourceforge.net/. [cited at p. 23]

[12] M. Ossmann, "Great scott gadgets." Retrieved 18 November 2012, from http://greatscottgadgets.com/. [cited at p. 23]

[13] Youtube, "Shmoocon 2011: Project ubertooth: Building a better bluetooth adapter." Retrieved 18 November 2012, from http://www.youtube.com/watch?v=KSd_1FE6z4Y. [cited at p. 23]

[14] B. SIG, "Bluetooth basics." Retrieved 05 Desember 2012, from http://www.bluetooth.com/Pages/low-energy.aspx. [cited at p. 23]

[15] M. Kershaw, "Kismet." Retrieved 10 Desember 2012, from http://www.kismetwireless.net. [cited at p. 23]

[16] Frontline, "Packet sniffers and protocol analyzers for bluetooth." Retrieved 16 Desember 2012, from http://www.fte.com/. [cited at p. 24]

[17] J. Padgette, "Bluetooth security in the dod," in *Military Communications Conference, 2009. MILCOM 2009. IEEE*, pp. 1 –6, oct. 2009. [cited at p. 54]

Appendices

Appendix A

measurements

Underneath follows the complete set of measurements done in the practical tests.

A.1 Range measurements

Meters	Test 1	Test 2	Test 3	Test 4	Test 5	Average
0	20	26	56	40	23	**33**
5	38	22	18	23	13	**22.8**
10	6	20	21	16	16	**15.8**
15	17	21	28	7	32	**21**
20	6	18	7	9	16	**11.2**
25	4	20	5	7	4	**8**
30	6	2	6	5	6	**5**
35	0	0	1	6	9	**3.2**
40	0	0	0	0	0	0
45	0	0	0	0	0	0
50	0	0	0	0	0	0
55	0	0	0	0	0	0

Table A.1: Packets captured with Antenna-1

Meters	Test 1	Test 2	Test 3	Test 4	Test 5	Average
0	78	54	94	75	87	**77.6**
5	34	79	25	68	28	**46.8**
10	22	27	24	21	64	**31.6**
15	35	23	34	58	26	**35.2**
20	20	31	17	17	40	**25**
25	9	8	10	11	10	**9.6**
30	0	0	16	3	1	**4**
35	26	7	11	25	27	**19.2**
40	6	0	12	0	7	**5**
45	2	18	27	0	4	**10.20**
50	0	0	0	0	0	0
55	0	0	0	0	0	0

Table A.2: Packets captured with Antenna-2