Defeating Bluetooth Low Energy 5 PRNG

for fun and jamming

Damien "*virtualabs*" Cauquil | DEF CON 27

digital.security

Who am I ?

- Security Evangelist @ digital.security
- Senior Security Researcher
- Main developer of **BtleJack** (a BLE swiss-army tool)

virtualabs

What’s new in BLE 5 ?

Bluetooth Low Energy Protocol

Origins

Bluetooth Low Energy has been introduced in 2010 as *Bluetooth Smart*, in Bluetooth Core Specifications **version 4.0**.

Evolved since then: version 4.1 (2013), version 4.2 (2014), version 5 (2016) and version 5.1 (2019)

Bluetooth Low Energy Protocol

Security mechanisms

- **Pairing**: key exchange with PIN code, OOB or ECDH
- **Secure connections**: encrypted and authenticated communication
- **Channel Selection Algorithm**: not a security mechanism, but makes sniffing complicated

Bluetooth Low Energy Protocol

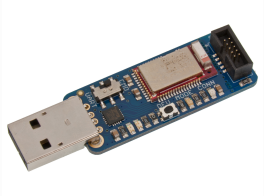
Known attacks (BLE <= 4.2)

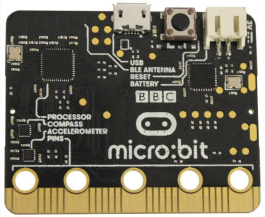
- **Sniffing**: stealing secrets sent through non-secure and secure BLE connections
- **Jamming**: disrupting an existing BLE connection
- **Hijacking**: taking over an existing BLE connection

Bluetooth Low Energy Protocol

Required hardware







Bluetooth Low Energy Protocol

BtleJack

- Swiss-army knife to **perform Bluetooth Low Energy attacks**
- Compatible with **Micro:Bit** and other **nRF51822** boards

New features introduced in BLE 5

- Better throughput (up to 2 Mbps)
- Better range (up to 800 feet – 240 meters)
- Improved coexistence

New features introduced in BLE 5

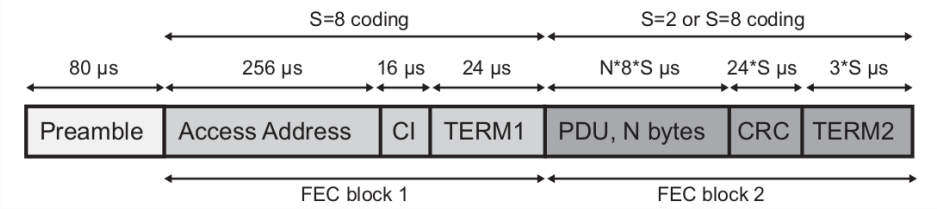
Better throughput / better range

BLE 5 adds two new PHYs:

- **2Mbps uncoded PHY**: twice the speed of BLE 4.x
- **LE Coded PHY**: range × 4 (125 kbps) or range × 2 (500 kbps)

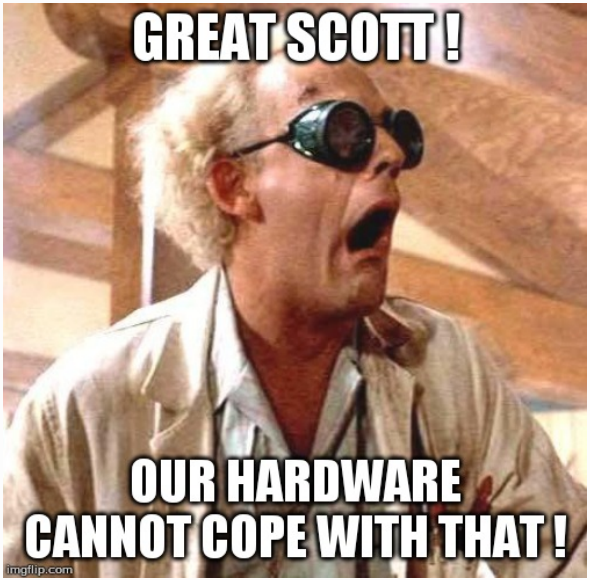
New features introduced in BLE 5

LE Coded PHY vs LE uncoded PHY



LSB MSB

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


GREAT SCOTT !
OUR HARDWARE
CANNOT COPE WITH THAT !
imgflip.com

New features introduced in BLE 5

Improved coexistence

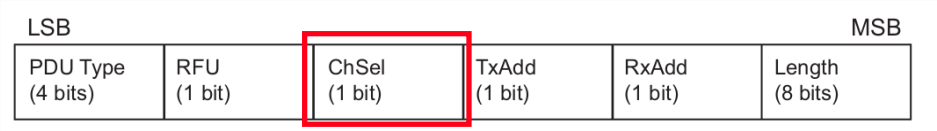
BLE 5 also adds a new **Channel Selection Algorithm** (**CSA #2**)

- Designed to replace the old algorithm:
  $Channel_{n+1} = (Channel_n + hopInc) \mod 37$
- **PRNG-based**: more ”random” than CSA #1

New features introduced in BLE 5

Improved coexistence

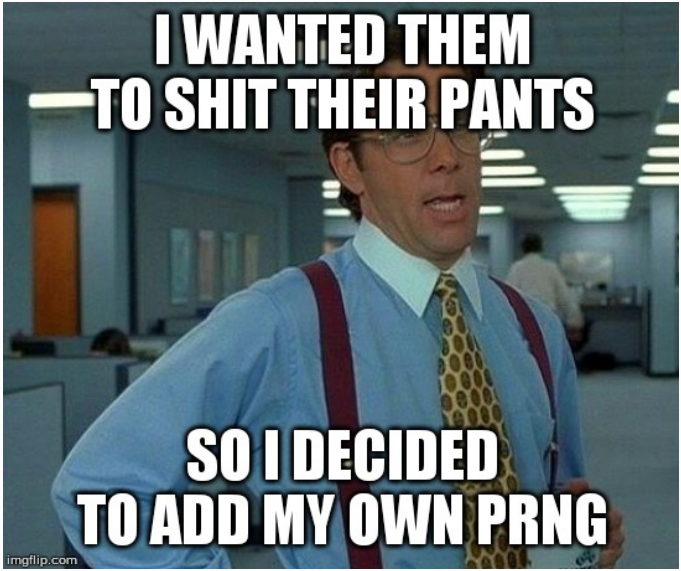
BLE 5 adds a **ChSel** bit in the advertising channel PDU header:



New features introduced in BLE 5

Consequences regarding known attacks

- A completely different hopping pattern (**65536-hop** instead of **37-hop** sequence) is used
- We won't be able to **sniff, jam or hijack** BLE 5 devices
- We need a **new hardware** that supports BLE 5 new PHYs


I WANTED THEM
TO SHIT THEIR PANTS
SO I DECIDED
TO ADD MY OWN PRNG
imgflip.com

BLE 5 PRNG WTF

BLE 5 PRNG overview

- It uses a **channel identifier** as a seed
- It relies on **basic operations** (add, multiply, xor, bit permutations)
- **Channel remapping** is performed if not all channels are in use

BLE 5 PRNG overview

Channel Identifier

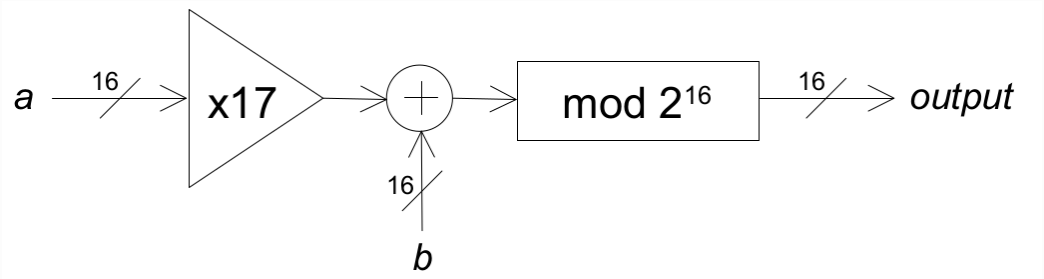
Channel identifier is first computed from **Access Address**, as described in the specs:

- The 16-bit input *channelIdentifier* is fixed for any given connection or periodic advertising; it is calculated from the Access Address by:
  channelIdentifier = (Access Address$_{31\text{-}16}$) XOR (Access Address$_{15\text{-}0}$)

BLE 5 PRNG overview

MAM

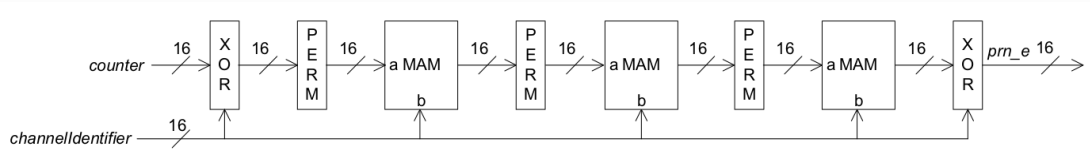
This PRNG relies on a basic routine called **MAM**:



BLE 5 PRNG overview

Random number generation

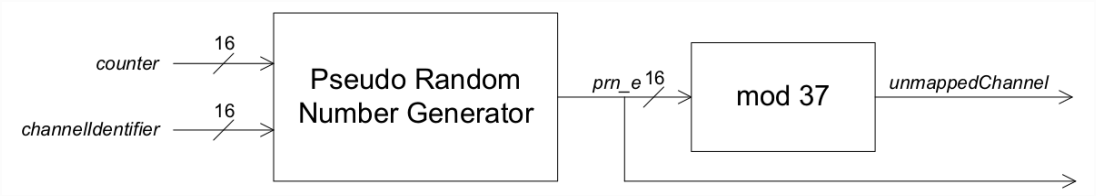
It generates a random number based on *channelIdentifier* and a *counter*:

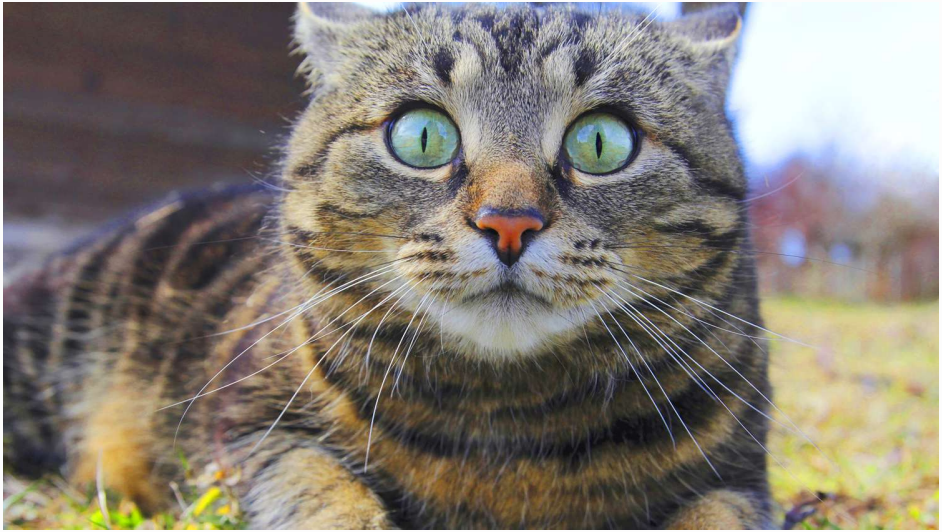


BLE 5 PRNG overview

Channel Selection Algorithm #2

Based on the output of this PRNG, it selects a specific channel:



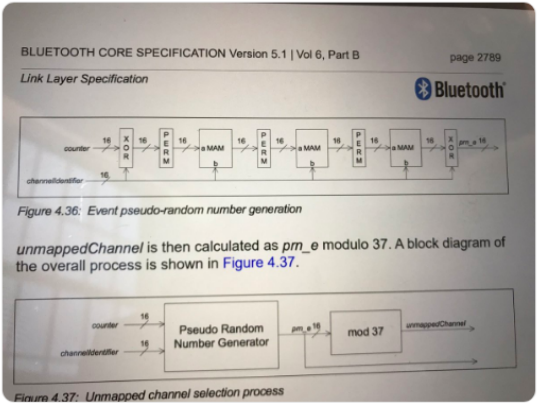


Matthew Green
@matthew_d_green

Suivre

Every single page of the Bluetooth spec should be illegal.

Traduire le Tweet



21:37 - 12 févr. 2019

Breaking this PRNG

I see some flaws in here …

- **channelIdentifier** is computed from **Access Address** … which is **PUBLIC** !

Breaking this PRNG

I see some flaws in here …

- **channelIdentifier** is computed from **Access Address** … which is **PUBLIC** !
- Next value is generated from a **COUNTER**, not from some previous internal state !

Breaking this PRNG

From 32 unknown bits to 16, easy peasy

Access Address is broadcasted in every packet!

- The 16-bit input *channelIdentifier* is fixed for any given connection or periodic advertising; it is calculated from the Access Address by:
  channelIdentifier = (Access Address$_{31\text{-}16}$) XOR (Access Address$_{15\text{-}0}$)

Breaking this PRNG

PRNG ? really ?

$$prn_e = PRNG(channelIdentifier, counter)$$

- **channelIdentifier is constant**
- it generates a fixed sequence of **65536 values**, indexed by **counter**


THAT'S NOT A PRNG
THAT'S A F*CKING SCRAMBLING FUNCTION
imgflip.com

Breaking this PRNG

Waht do we need to break it ?

- Consider **channelIdentifier as known**

Breaking this PRNG

Waht do we need to break it ?

- Consider **channelIdentifier as known**
- We are left with an **unknown 16-bit counter**

Breaking this PRNG

Waht do we need to break it ?

- Consider **channelIdentifier as known**
- We are left with an **unknown 16-bit counter**
- If we can figure out **where we are** in the sequence of **65536 values**, we would find out this counter value

Guessing the counter value

Getting information from RF

As an attacker, we can only monitor an existing BLE connection and:

- determine **WHEN** a packet is sent over a specific channel
- determine the **time spent between two packets** transmitted on two channels

Getting information from RF

Channels are not 16-bit values

If all 37 data channels are used:

$$channel = prn_e \mod 37$$

else:

$$channel = remappingIndex[\frac{(N \times prn_e)}{2^{16}}]$$

First approach: using a sieve

- **Pre-requisite**: we know the **hopInterval** used for the target connection (i.e. time spent between two hops)
- Our goal: to **eliminate** every possible **candidate** in the smallest number of rounds

First approach: using a sieve

How it works

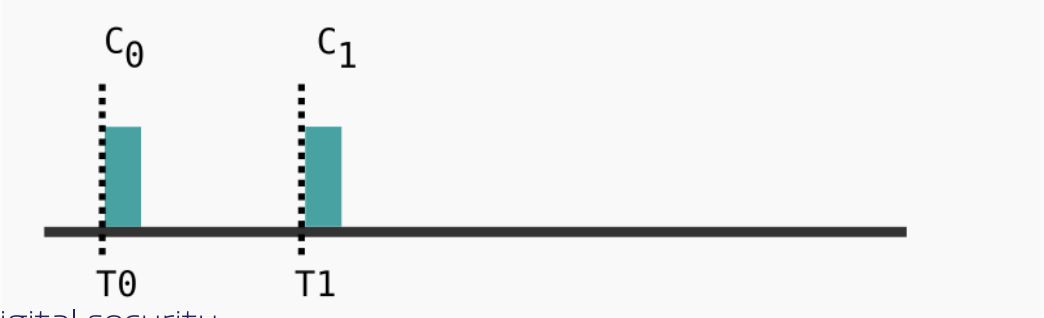
Considering a counter of 0, we compute channel C_0 from PRNG with this candidate counter, and wait for a valid packet



First approach: using a sieve

How it works

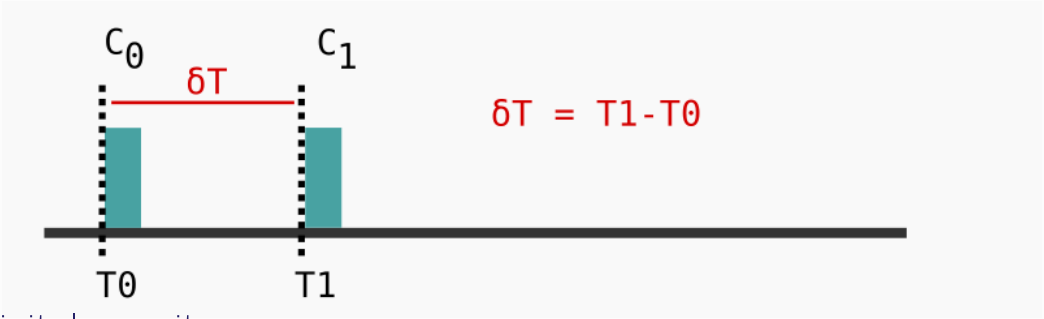
Once a valid packet received, we compute the next channel (C_1) and we wait for a valid packet again:



First approach: using a sieve

How it works

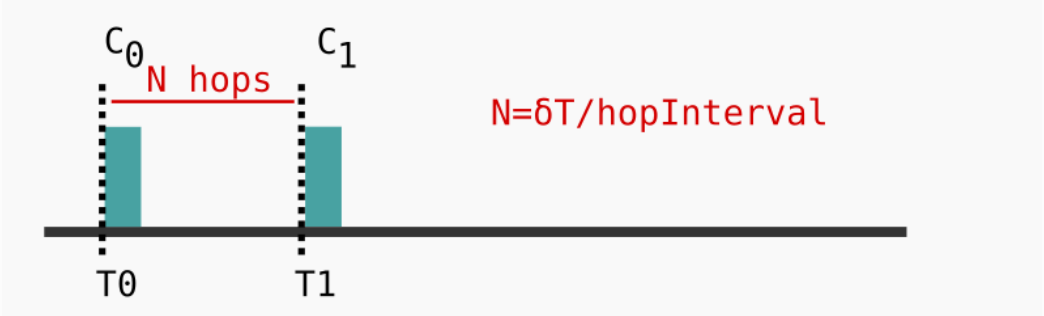
We deduce the time spent between these two packets:



First approach: using a sieve

How it works

And we convert it into a number of hops:



First approach: using a sieve

How it works

We then sieve the sequence of channels to only keep potential candidate indexes:

$$Sequence = \{\ldots, \underbrace{C_0, \ldots, C_1}_{\text{N hops}}, \ldots\}$$

And we end up with a **list of candidate counter values** at *T0* (between 200 and 400, most of the time)

First approach: using a sieve

How it works

Then, we consider the first candidate and:

- we compute C_0 and C_1

First approach: using a sieve

How it works

Then, we consider the first candidate and:

- we compute C_0 and C_1
- we wait for a packet on channel C_0

First approach: using a sieve

How it works

Then, we consider the first candidate and:

- we compute C_0 and C_1
- we wait for a packet on channel C_0
- we wait for another packet on channel C_1

First approach: using a sieve

How it works

Then, we consider the first candidate and:

- we compute C_0 and C_1
- we wait for a packet on channel C_0
- we wait for another packet on channel C_1
- we deduce the number of hops

First approach: using a sieve

How it works

Then, we consider the first candidate and:

- we compute C_0 and C_1
- we wait for a packet on channel C_0
- we wait for another packet on channel C_1
- we deduce the number of hops
- we sieve our list of candidates to only keep the matching ones

First approach: using a sieve

How it works

Repeat until **one candidate** left!

(This is the counter value at *T0*)

First approach: using a sieve

Implementation (video)

Since I did not have a BLE 5 compatible device, I simulated this attack:

```
-> Attacker waited 21 hops until a packet arrived on channel 0
>> Filtering candidates ...
-> Candidate 20476 removed
-> Candidate 45108 removed
-> Remaining candidates: 22967
>> Found initial counter: 22967
INFO: Attack required 209 hops
```



Guessing the hop interval

Guessing the hop interval

Using our previous measures

$$hopInterval = GCD(\delta T_0, \delta T_1, \delta T_2, \ldots)$$

Guessing the hop interval

Simulating (again)

```
$ python3 csa2-hopinter-simulate.py
[system] Hop interval: 2220
Deduced hop interval after 2 deltas: 8880
Deduced hop interval after 3 deltas: 4440
Deduced hop interval after 4 deltas: 4440
Deduced hop interval after 5 deltas: 2220
Deduced hop interval after 6 deltas: 2220
[...]
Deduced hop interval after 10 deltas: 2220
```

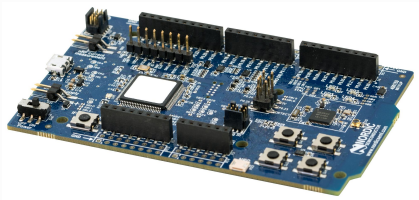
Guessing the hop interval

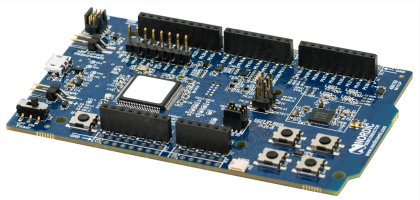
Efficiency

# of measures	approx. success rate	Max. time required (s)
5	95%	124
6	98%	119
7	99%	134
8	99%	165
9	99%	183
10	100%	178

From theory to practice: breaking CSA #2 on the field

I bought some BLE 5 compatible devices





Improving Btlejack

Channel map and hopInterval recovery

- Btlejack can do the job but only for **1 Mbps uncoded PHY**
- I modified **Btlejack** to compute **hopInterval** while mapping channels
- This GCD-based technique works pretty well =)

Improving Btlejack

Example output

```
# btlejack -f 0x9af4493f -5 -d /dev/ttyACM2
BtleJack version 2.0

[i] Using cached parameters (created on 2019-07-23 00:47:59)
[i] Detected sniffers:
> Sniffer #0: fw version 2.0
[i] Synchronizing with connection 0x9af4493f ...
✓ CRCInit: 0x6f1c40
✓ Channel Map = 0x0774ea8a3c
✓ Hop interval = 160
```


BORD
ORDERS

Improving Btlejack

Implementing our sieve attack on counter

I implemented this algorithm and tested it:

- First round **went pretty well**: I got about 250 candidates filtered

Improving Btlejack

Implementing our sieve attack on counter

I implemented this algorithm and tested it:

- First round **went pretty well**: I got about 250 candidates filtered
- Next rounds messed up: I **was not able to guess** the counter value

Improving Btlejack

Implementing our sieve attack on counter

Filter out candidates took **a hell of a time**, thus inducing a delay !

(Sorting algorithms 101 did not help)



Improving Btlejack

Re-designing our sieve attack

To solve this problem, I moved from a **sieve** to a **pattern-matching algorithm**:

- **Measure first** (number of hops between 11 different consecutive channels)

Improving Btlejack

Re-designing our sieve attack

To solve this problem, I moved from a **sieve** to a **pattern-matching algorithm**:

- **Measure first** (number of hops between 11 different consecutive channels)
- You end up with **10** δT **values** (δT_0 to δT_9)

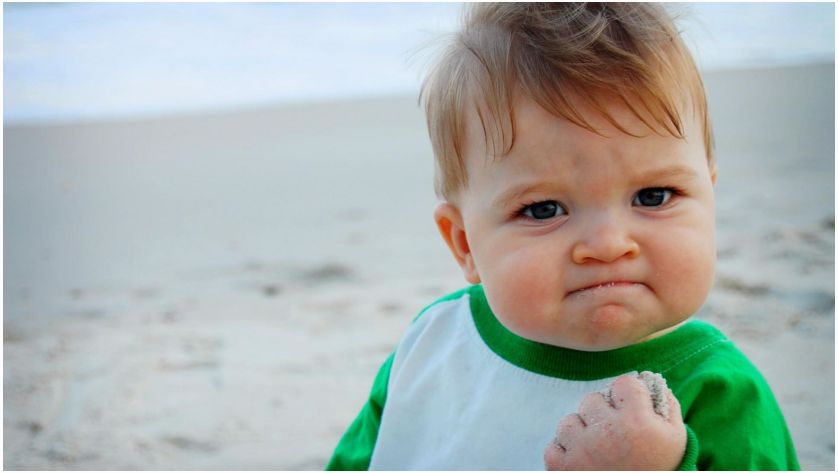
Improving Btlejack

Re-designing our sieve attack

To solve this problem, I moved from a **sieve** to a **pattern-matching algorithm**:

- **Measure first** (number of hops between 11 different consecutive channels)
- You end up with **10** δT **values** (δT_0 to δT_9)
- Look for this **hopping pattern** in our sequence and **deduce counter**

counter guessing is now working, YAY !




GUESS HOP INTERVAL
GUESS "PRNG" COUNTER
SYNC
SYNC
imgflip.com

Improving Btlejack

13 hops why

Pattern matching took about **13 hops** to complete !

Sniffing a BLE 5 connection with CSA #2

```
[i] Synchronizing with connection 0x31204f95 ...
✓ CRCInit: 0x40d64f
✓ Channel Map = 0x1fffffffff
✓ Hop interval = 160
✓ CSA2 PRNG counter = 5137
[i] Synchronized, packet capture in progress ...
LL Data: 0e 08 04 00 04 00 52 10 00 01
LL Data: 02 08 04 00 04 00 52 10 00 00
LL Data: 02 08 04 00 04 00 52 10 00 01
LL Data: 0e 08 04 00 04 00 52 10 00 00
LL Data: 0e 08 04 00 04 00 52 10 00 01
```

Jamming a BLE 5 connection with CSA #2

```
# btlejack -f 0x91a7471f -m 0x1fffffffff -p 160 -5 -j
BtleJack version 2.0
[..]

[i] Synchronizing with connection 0x91a7471f ...
✓ CRCInit: 0x21a31e
✓ Channel map is provided: 0x1fffffffff
✓ Hop interval is provided: 160
✓ CSA2 PRNG counter = 1199
[i] Synchronized, jamming in progress ...
[!] Connection lost.
[i] Quitting
```

Btlejack version 2.0

https://github.com/virtualabs/btlejack

Conclusion

Conclusion

Bluetooth Low Energy 5 CSA #2 PRNG **is weak**:

- It is not really a **PRNG**
- **16 bits** out of 32 **can be easily guessed** from a public value
- The last 16 bits compose **a counter** that is periodically incremented

Conclusion

This PRNG:

- is thought to **improve coexistence, not security**
- is designed **to be easily implemented** on low-power devices

Conclusion

- We should use Nordic's nRF52840 rather than the old nRF51822
- Still a lot of work to port **BtleJack** to this platform
- But luckily, **Marcus Meng** has done a great job with nRF52840

Research materials

Python scripts and notes

`https://github.com/virtualabs/ble5-research`

Thank you !

Questions ?

@virtualabs

damien.cauquil@digital.security

Further readings

- https://virtualabs.fr/defcon26/DC26-DamienCauquil-YoudBetterSecureYourBLEDevices.pdf
- https://www.alchemistowl.org/pocorgtfo/pocorgtfo17.pdf
- https://github.com/mame82/misc/blob/master/logitech_vuln_summary.md