



Analyzing the security of a Fitbit Wearable

Bechir BOUALI & Baptiste NEVEU

Advisors: Aurélien FRANCILLON & Marius MUENCH

June 26, 2019

Abstract

Fitness trackers are more and more used in our everyday life. They record sensitive informative, such as step counts, heart rate, etc. In that way, their security is an important part of their design. Previous studies had shown that some companies took this subject seriously, while other did not.

Tables of contents

Abstract	1
Introduction	3
Related work	3
Fitbit Hardware access	4
Reach the Printed Circuit Board (PCB)	4
Connection to the fitbit	5
Setup connection between the debug adapter and the board	5
Configuration of the debugger software	6
Dump the firmware	7
Static analysis of the firmware	9
Dynamic analysis of the firmware	10
Manually debugging: GDB	10
Analyzing firmware using Avatar ²	13
Avatar ² basics	13
Breakpoint Avatar ² script	13
Results	14
Reaching breakpoints	14
Testing Bluetooth function in the emulator	14
Issues encountered with Avatar	16
Further work	16
Conclusion	16
Table of references	17
Tables of figures	18
Annexes	19

Introduction

Fitbit has many fitness trackers. Among them, we can quote the Fitbit Flex. This tracker is one of the first fitness tracker introduced in 2013. The previous studies stated that even if the security aspect was taken into consideration, the Fitbit Flex architecture was not designed with security in mind. This project has multiples objectives:

- Disassemble the Fitbit and extract the chip
- Communicate with the chip with the official application
- Use debugging tools available thanks to OpenOCD, such as GDB
- Dump the firmware
- Analyse it
- Use the Fitbit with Avatar², the target orchestration framework with focus on dynamic analysis of embedded devices' firmware

Related work

At first, we looked at the work by Classen et al.^[1]. In this paper, they explain how they proceeded to analyse the security of the Fitbit and on which firmware version they did it. This paper described a debugger option in the Fitbit application on Android. We briefly tried to enable this mode by decompiling the APK and by modifying the bytecode in the specified file. After that, we recompile the APK and check the result; it did not work, may be due to a malformed table in the Android bytecode. After that, we wanted to disassemble the Fitbit, we rely on the previous work made by Jiska Classen et al. This paper describes the security of the Fitbit Cloud, App and firmware.

1. Fitbit Hardware access

1.1. *Reach the Printed Circuit Board (PCB)*

To reach the PCB, the plastic part that covers it needs to be melted. We use a dryer to do it. As the Bluetooth antenna is located on the top of the Fitbit, which formed a solid part with the LED, an extra attention has to be taken into account. To solder the pins, it is very easy to remove a part of the PCB that even if tiny, is important. At the end, the PCB looks like this:

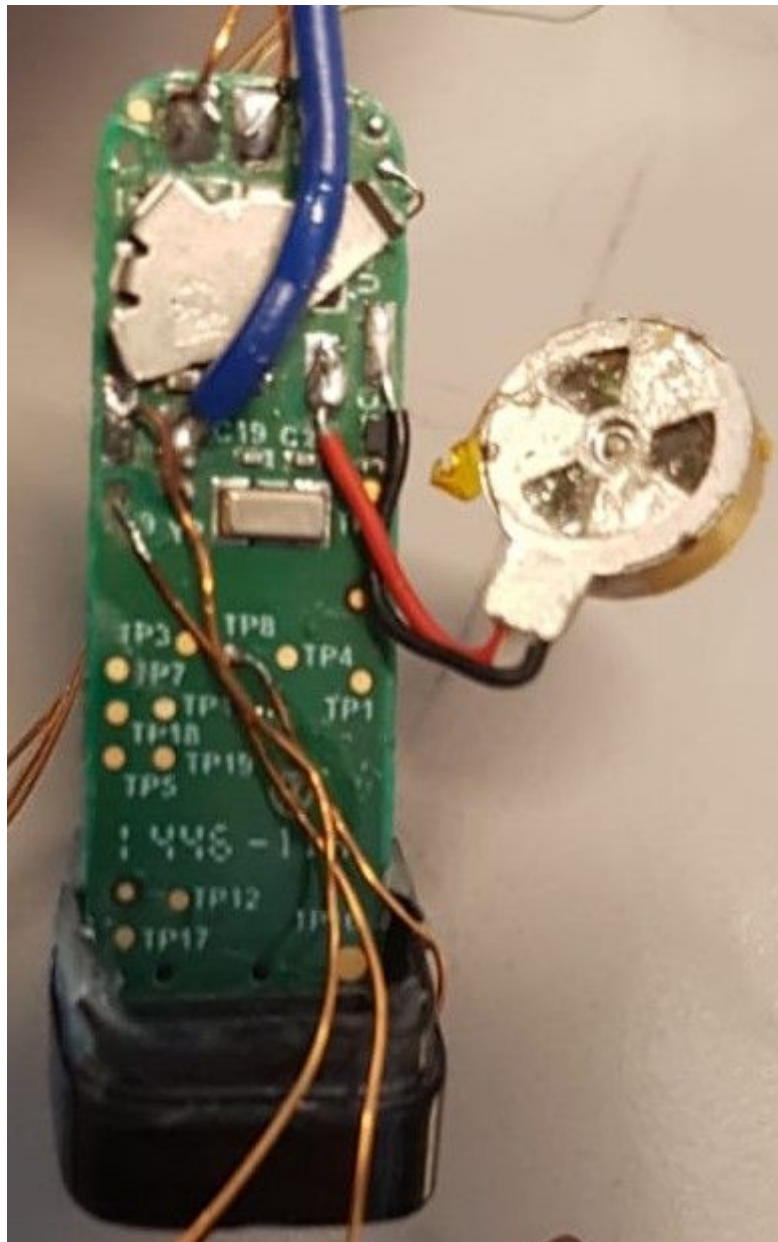


Figure 1: PCB of a modified fitbit tracker with solder connections

1.2. Connection to the fitbit

According to related work^[2] the fitbit could be connected to a Debug unit through an Serial Wire Debug (SWD) interface and on circuit we found four test points that could be used for debugging.

1.2.1. Setup connection between the debug adapter and the board

As a debug adapter we will use STLINK-V2-1 of a nucleo board that support SWD. According to the datasheet^[3] the SWD pins are like this:

- SWDCLK: Pin 2
- GND : Pin 3
- SWDIO: Pin 4
- NRST: Pin 5

Here we need to insist about the importance of the reset pin which will allow the debugging when connecting to the chip otherwise the firmware will disable the debug interface^[4].

The matching between ST-LINK-V2-1 and the Fitbit test points should be as follow:

SWD pins	ST-LINK-V2-1 pins	Fitbit test points
SWDCLK	Pin 2	TP8
SWDIO	Pin 4	TP9
GND	Pin 3	GND
NRST	Pin 5	TP10

Table 1: Pairing between ST-LINK-V2 and fitbit

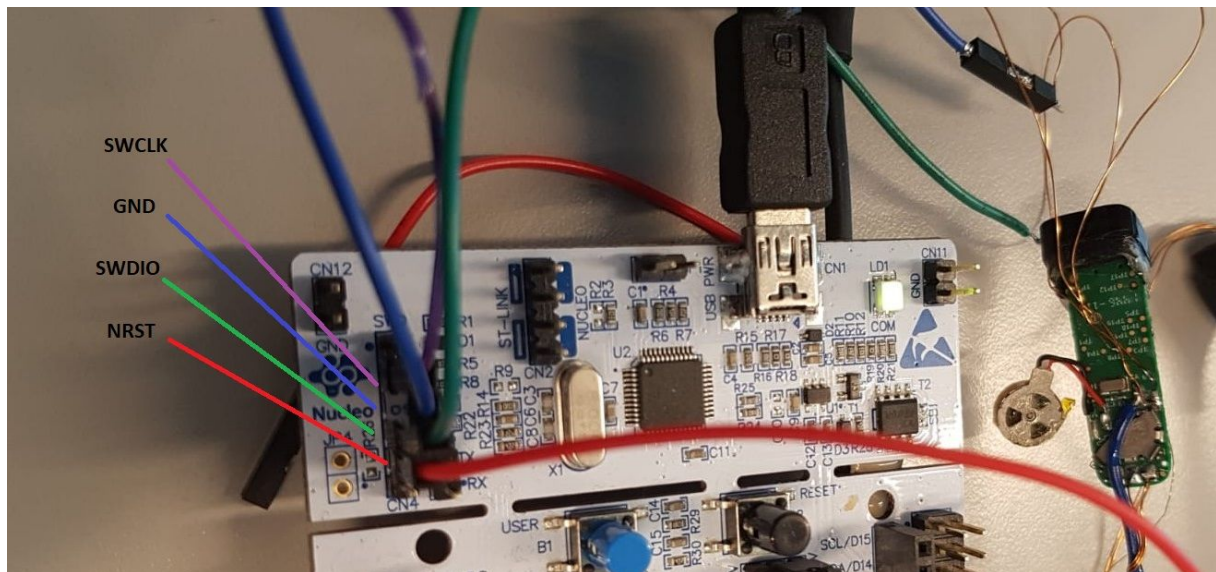


Figure 2: Connection between the Nucleo board and the Fitbit

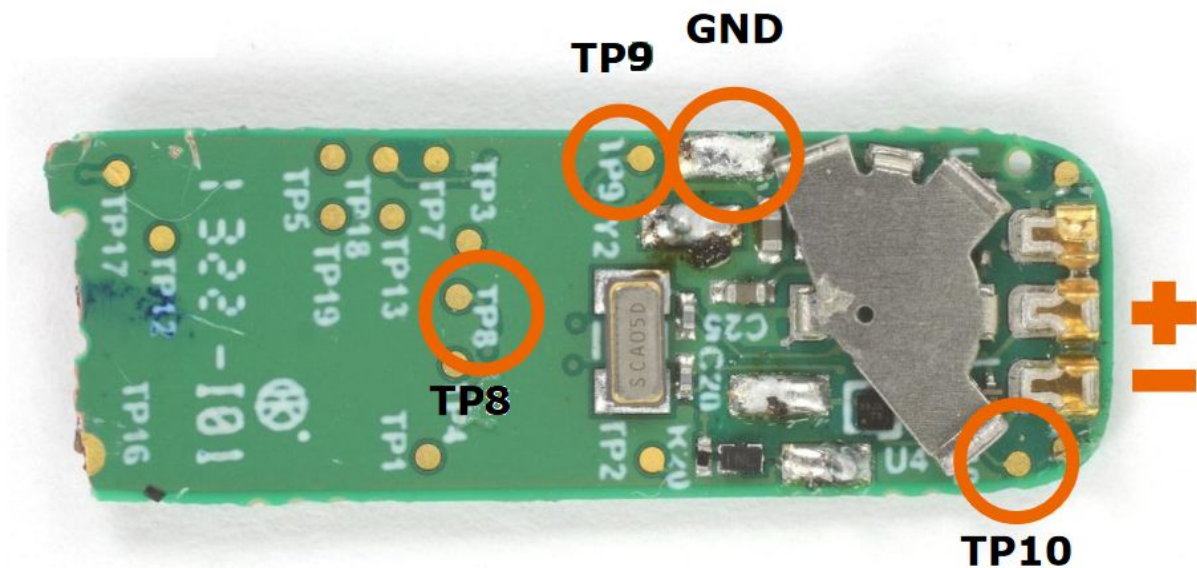


Figure 3: Fitbit with the required test points, from the Recon talk by Classen et al.^[2]

1.2.2. Configuration of the debugger software

As a debugger software we choose to work with openOCD^[5]. It is an open on-chip debugging in-system programming and boundary-scan testing tool for ARM and MIPS systems that offers the possibility to interact with the tracker. In our case the target is STM32L151UC (ARM Cortex M3).

We prepared an openOCD configuration file (see annexes) as follow:

- The content of stm32l1.cfg.
- The content of stlink-v2.cfg: configuration of the debugger.
- We added “transport select hla_swd”: to specify that we will use SWD.

- We added “reset_config srst_only srst_nogate connect_assert_srst” : to enable hardware reset.
- We added “reset_config none separate”: to enable the reset to be done internally over the SWD channel which will avoid errors while debugging using GDB.

But we also tried the ST-LINK Utility^[6] software to interact with the chip. The connection required extra settings that will be detailed later.

1.3. Dump the firmware

Performing an acquisition of the firmware image is crucial as it allows us to analyze the firmware and to do static analysis in order to find interesting functions. To dump the firmware, we proceeded 2 different ways:

- Using openOCD:
 - “openocd -f fitbit.cfg”: launch openocd process.
 - “telnet 127.0.0.1 4444”: connect to openocd interface.
 - “dump_image firmware.bin 0x0 0x40000”: dump the firmware and store the data in file named firmware.bin.
- Using ST-LINK Utility:

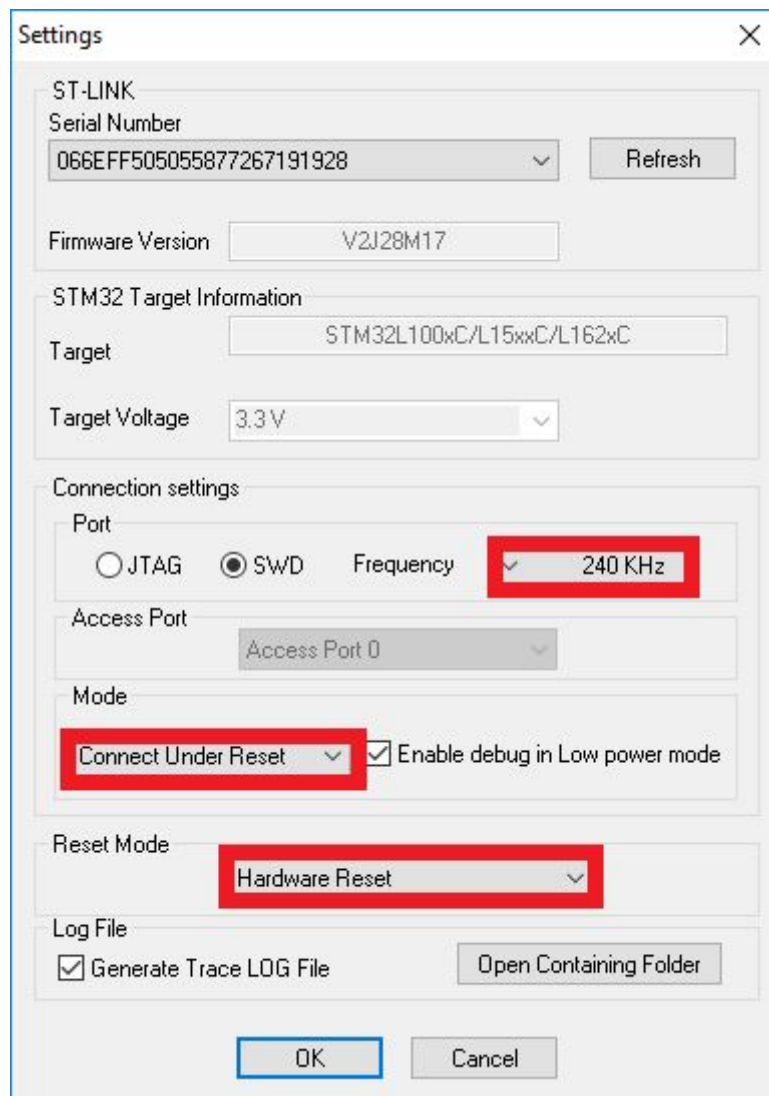


Figure 4 : Settings to connect to the Fitbit using ST-LINK Utility

After these settings, we are now able to access to the firmware image and to dump it. We got the same output as with the openOCD way.

2. Static analysis of the firmware

On this part, we will develop how we proceed to do static analysis of the dumped firmware. First, we looked at the assembly code in order to understand how it works.

We use IDA Pro to disassemble the firmware. As the disassembly code is too big to analyze, we will not focus on the wholeness of it. Instead, we will focus on identifying functions that process Bluetooth packets.

The library in the firmware used for Bluetooth packets is almost identical to an open-source library used for an Arduino BLE Breakout Board. Thanks to an IDC script shared by the team that worked on a similar project, we were able to extract useful information about the bluetooth functions.

- 0x0800EE62: get_bluetooth_id.

```
App:0800EE62 ; signed int __fastcall get_bluetooth_id(int mac_addr_dst, unsigned __int8 mac_addr_len)
App:0800EE62 get_bluetooth_id ; CODE XREF: send_bt_id_c0+2A+P
```

Figure 5: Get_bluetooth_id function

- 0x08012C44: exti_bluetooth_record.
- 0x08018868: rf_record_bluetooth.
- 0x080214A0: printf_bluetooth_id.
- The bluetooth stack BSL is located at 0x080036E0.

To check if the previous addresses really match the desired functions, we can check this with dynamic analysis.

3. Dynamic analysis of the firmware

3.1. *Manually debugging: GDB*

GNU Project Debugger (GDB) is a debugger that allows users to see what is going inside their programs, e.g. to trace and alter their execution^[7]. The user can monitor the programs or do remote debugging.

We need first to execute the openOCD which will run a gdbserver locally on port 3333 and we connect to it as follow:

```
#run openocd
$ openocd -f fitbit.cfg
#run gdb
$ arm-none-eabi-gdb
(gdb) target extended-remote 127.0.0.1:3333
```

In fact, when you try to do remote GDB debugging with the default firmware, the device will crash because when continuing execution, the Fitbit firmware reconfigures GPIO pins (GPIO pins 13 and 14 are assigned to SWCLK and SWDIO respectively, which enables debugging) to analog mode, thereby disabling debugging.

That is why the firmware need to be patched in order to allow them. To flash the firmware, there is two ways to do it:

- The first one is to download the Android application developed by the same team following this repository^[8]. By doing so, it also enables nice features, such as scanning for Bluetooth Low Energy Fitbit devices, dump memory, etc. With this, we were able to dump the firmware on the Android device. The firmware flashing is done by the application by bluetooth. However, everytime we tried it, we got a connection timeout and we were never able to flash it.
- The second one is by flashing the firmware using openocd. There are few steps required to do it:

```
> init
> reset init
> halt
> flash write_image erase firmware.bin 0x08000000
> reset run
```

In the new firmware, there is a GDB backdoor added to enable debugging and that by reconfigure the functionalities of the two pins TP8 and TP9. So this will avoid this following error while debugging : " jtag status contains invalid mode value - communication failure"

Once done, we are able to debug the fitbit wearable with GDB. The most interesting with it is to set breakpoints in order to find some functions. The main drawback about flashing this firmware is that it causes a desynchronization with the Fitbit server.

As stated in the previous part, we set breakpoint for four addresses to check if they correspond:

- 0x0800EE62: get_bluetooth_id
- 0x08012C44: exti_bluetooth_record
- 0x08018868: rf_record_bluetooth
- 0x080214A0: printf_bluetooth_id

Using GDB we set hardware breakpoints :

```
$arm-none-eabi-gdb
$gdb source -s gdb-conf
$gdb monitor halt
$gdb hb *0x0800EE62
$gdb hb *0x08012C44
$gdb hb *0x08018868
$gdb hb *0x080214A0
$gdb continue
```

`gdb-conf` is gdb configuration file contain this lines:

```
target extended-remote 127.0.0.1:3333
set remote hardware-breakpoint-limit 6
```

```
set remote hardware-watchpoint-limit 4  
  
monitor halt
```

After that, we try to connect the smartphone with the fitbit using bluetooth using the pairing function available in the Fitbit app:

```
gdb-peda$ info b  
Num      Type      Disp Enb Address      What  
1        hw breakpoint keep y  0x0800ee62  
3        hw breakpoint keep y  0x08018868  
4        hw breakpoint keep y  0x080214a0  
gdb-peda$ c  
Continuing.  
  
Breakpoint 1, 0x0800ee62 in ?? ()
```

Figure 6: Reaching the breakpoint with GDB

The first part of the capture shows that the breakpoints were correctly set. The next part means that the first breakpoint has been reached, and according to the output on the smartphone, it does show that this address is used for bluetooth pairing.

The next step is now to reproduce this whole process; from setting breakpoints to reach them, using Avatar², the target orchestration framework with focus on dynamic analysis of embedded devices' firmware^[9].

3.2. Analyzing firmware using Avatar²

3.2.1. Avatar² basics

Avatar²^[9] is a target orchestration framework with focus on dynamic analysis of embedded devices' firmware. To use Avatar², you need three parts:

- The target that will be used to perform and analyze the firmware code. For this project, we emulate the fitbit tracker with Qemu (which is a generic and open source machine emulator and virtualizer), and we use the Fitbit when needed, for example when an execution need to be performed with some components that can not be emulated.
- The memory layout needs to be configure in order to know where the memory address are beginning and ending. For this project, we set the memory range for the ROM, the RAM and the MMIO.
- The execution plan describes how Avatar will be executed between the multiples targets.

3.2.2. Breakpoint Avatar² script

The first part of this script is to initialize the configuration, to tell where the firmware, the openOCD configuration file and the QEMU path are located. After that, we create an Avatar object and we set up its architecture, an openOCD object, a QEMU object and we specify the memory ranges for ROM, RAM and MMIO. Once everything is done, we can initialize the avatar object. After that, we set the breakpoint at the address 0x0800EE62 which corresponds to the `get_bluetooth_id` function. We then continue the process, and wait to reach this breakpoint. When the program counter is at this address, the execution will be transferred to QEMU which allows further analysis.

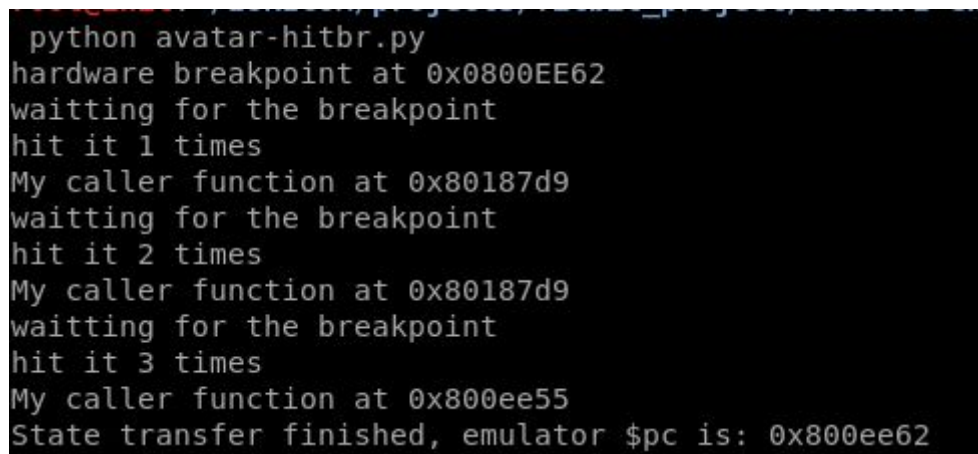
To make this script working, we ran into multiples issues that will be detailed after.

3.2.3. Results

3.2.3.1. Reaching breakpoints

Using avatar script we were able to reach the breakpoint at 0x0800EE62: get_bluetooth_id when we use the mobile application and try to pair the fitbit with an Android device. But before that, this address was reached twice in the initialization process .

- First & second hit: get_bluetooth_id was called by "0x80187d9" instruction inside the "bluetooth_record" function.
- Third hit: when we try to pair the app and the fitbit. Here a function 0x800ee55: "send_bt_id_c0" call get_bluetooth_id and then send bluetooth_id to the app which will enable the pairing.



```
python avatar-hitbr.py
hardware breakpoint at 0x0800EE62
waitting for the breakpoint
hit it 1 times
My caller function at 0x80187d9
waitting for the breakpoint
hit it 2 times
My caller function at 0x80187d9
waitting for the breakpoint
hit it 3 times
My caller function at 0x800ee55
State transfer finished, emulator $pc is: 0x800ee62
```

Figure 7: Avatar breakpoint results

3.2.3.2. Testing Bluetooth function in the emulator

After reaching the third breakpoint the state was transferred to Qemu to emulate the execution. So by using lpython.embed(), we were able to continue testing other functionalities and reaching other breakpoints while using the emulator.

```

In [16]: fitbit.set_breakpoint(0x08012C44,hardware=True)
Out[16]: 2

In [17]: fitbit.cont()
Out[17]: True

In [18]: fitbit.get_status()
Out[18]: {'state': <TargetStates.STOPPED: 4>}

In [19]: fitbit.regs.pc
Out[19]: 134294596

In [20]: hex(134294596)
Out[20]: '0x8012c44'

```

Figure 8: Output when reaching the second breakpoint

In addition we were able to get the MAC address of the fitbit by debugging the bluetooth function that use a MAC address as a parameter. With the help of the disassembly code, we found that each time the function `get_bluetooth_id` is called, it load in the register R1 the address of memory where the MAC address is stored.

```

In [28]: hex(fitbit.regs.pc)
Out[28]: '0x800ee6e'

In [29]: hex(fitbit.regs.r1)
Out[29]: '0x200049f0'

In [30]: hex(fitbit.read_memory(0x200049f0,8,1,False))
Out[30]: '0xccd1fa829b03'

In [31]: 

```

Figure 9: Value of the register R1

```

p:0800EE62 ; signed int __fastcall get_bluetooth_id(int mac_addr_dst, unsigned __int8 mac_addr_len)
p:0800EE62 get_bluetooth_id ; CODE XREF: send_bt_id_c0+2A1p
p:0800EE62 ; bluetooth_record_sth+2C1p ...
p:0800EE62 PUSH {R7,LR} ; returns bluetooth id or r0=0 if error
p:0800EE64 UXTB R1, R1 ; Unsigned extend byte to word
p:0800EE66 CMP R1, #6 ; Set cond. codes on Op1 - Op2
p:0800EE68 BCC loc_800EE76 ; Branch
p:0800EE6A MOVS R2, #6 ; length
p:0800EE6C LDR R1, =bt_mac_address ; Load from Memory
p:0800EE6E BL memcpy_wrapper ; Branch with Link
p:0800EE72 MOVS R0, #0 ; Rd = Op2
p:0800EE74 B locret_800EE78 ; Branch
p:0800EE76 ;

```

Figure 10: Load from memory

We can also see the fitbit MAC address with the figure 9: cc:d1:fa:82:9b:03

3.2.4. Issues encountered with Avatar²

At first, to set up the breakpoints with Avatar², we tried to transfer the control to QEMU. However the execution stop before reaching the breakpoint due to multiple errors:

- "Undefined debug reason 7 ": the OpenOCD doesn't know which state that the MCU is in at the moment. So we need a halt there.
- "Unable to create rx_queue" : The problem here is that when we create the qemu target we used the normal qemu binary "qemu-system-arm" but we should employ qemu with avatar extension^[10].

4. Further work

To better analyze the bluetooth functions we can utilize avatar-panda^[11] module to record an execution of the fitbit while connecting to the mobile application and then reply the recorded execution with focus on various analysis tasks during the reply.

Conclusion

During this project, we worked on many fields, from the hardware part when we had to disassemble the fitbit tracker to get access to the PCB, to the dynamic analysis of the fitbit. We were also able to find and test bluetooth functions.

- Static analysis by disassembling the firmware using IDA Pro.
- Dynamic analysis by using Avatar and the ability to transfer the execution to an emulator.

Table of references

- [1] Anatomy of a Vulnerable Fitness Tracking System:Dissecting the Fitbit Cloud, App, and Firmware, <http://homepages.inf.ed.ac.uk/ppatras/pub/imwut18.pdf>
- [2] Fitbit Hacking - RECON,
https://recon.cx/2018/montreal/schedule/system/event_attachments/attachments/000/000/045/original/RECON-MTL-2018-Fitbit_Firmware_Hacking.pdf
- [3] STM32 Nucleo-64 boards User manual,
https://www.st.com/content/ccc/resource/technical/document/user_manual/98/2e/fa/4b/e0/82/43/b7/DM00105823.pdf/files/DM00105823.pdf/jcr:content/translations/en.DM00105823.pdf
- [4] OpenOCD reset configuration, <http://openocd.org/doc/html/Reset-Configuration.html>
- [5] OpenOCD website, <http://openocd.org/>
- [6] ST-LINK utility software, <https://www.st.com/en/development-tools/stsw-link004.html>
- [7] GDB website, <https://www.gnu.org/software/gdb>
- [8] Fitbit Open Source Android App,
<https://github.com/seemoo-lab/fitness-app/tree/master/release>
- [9] Avatar repository, <https://github.com/avatartwo/avatar2>
- [10] Avatar-qemu, <https://github.com/avatartwo/avatar-qemu>
- [11] PANDA backend for avatar², <https://github.com/avatartwo/avatar-panda>

Tables of figures

Figure 1: Figure 1: PCB of a modified fitbit tracker with solder connections, p4

Figure 2: Connection between the Nucleo board and the Fitbit, p6

Figure 3: Fitbit with the required test points, CC BY-NC-SA 3.0 ifixit.com (Sam Lionheart), p6

Figure 4 : Settings to connect to the Fitbit using ST-LINK Utility, p8

Figure 5: Get_bluetooth_id function, p9

Figure 6: Reaching the breakpoint with GDB, p12

Figure 7: Avatar breakpoint results, p14

Figure 8: Output when reaching the second breakpoint, p15

Figure 9: Value of the register R1, p15

Figure 10: Load from memory, p15

Annexes

avatar script to reach breakpoint and to transfer execution on emulator

```
import IPython
import telnetlib
import time
from os.path import abspath
from avatar2 import *
def main():
    # Configure the location of various files
    firmware = abspath('./accelerometer_firmware.bin')
    openocd_config = abspath('./fitbit.cfg')
    qemu_path =
abspath("/root/EURECOM/projects/fitbit_project/avatar2/build/arm-sofmmmu/qemu-syst
em-arm")
    # Initiate the avatar-object
    avatar = Avatar(arch=ARM_CORTEX_M3, output_directory='/tmp/avatar')
    # create openocd object and enabling gdbserver on port 3333
    fitbit =
avatar.add_target(OpenOCDTarget,openocd_script=openocd_config,gdb_executable="ar
m-none-eabi-gdb")
    # create qemu object to emulate
    qemu = avatar.add_target(QemuTarget,
executable=qemu_path,gdb_executable="arm-none-eabi-gdb", gdb_port=1236)
    #Define the various memory ranges and store references to them
    rom = avatar.add_memory_range(0x08000000, 0x40000, file=firmware)
    ram = avatar.add_memory_range(0x20000000, 0x8000)
```

```

        mmio = avatar.add_memory_range(0x40000000, 0x1000000,forwarded=True,
forwarded_to=fitbit)

    try:

        avatar.init_targets()

        # set hardware breakpoint at 0x0800EE62(get_bluetooth_id)

        fitbit.set_breakpoint(0x800ee62, hardware=True)

        print ("hardware breakpoint at 0x0800EE62")

        fitbit.cont()

        n = 0

        while True:

            time.sleep(3)

            fitbit.wait()

            if fitbit.regs.pc == 0x0800EE62:

                n += 1

                print ("hit it %d times" % n)

                print ("My caller function at 0x%x" % fitbit.regs.lr)

            if n == 3:

                break

            print("waitting for the breakpoint")

            fitbit.cont()

        #Transfer the state from the physical device to the emulator

        avatar.transfer_state(fitbit, qemu, sync_regs=True,synced_ranges=[ram])

        print("State transfer finished, emulator $pc is: 0x%x" % qemu.regs.pc)

    except:

        print("error somewhere, shutdown avatar ")

        #shutting down avatar cleanly if there is an error

        avatar.shutdown()

    qemu.cont()

```

Further analysis could go here:

```
IPython.embed()
```

```
time.sleep(5)
```

```
print("The End ,shutdown avatar")
```

```
avatar.shutdown()
```

```
if __name__ == '__main__':
```

```
    main()
```

OpenOCD configuration file:

```
#
# STMicroelectronics ST-LINK/V2-1 in-circuit debugger/programmer
#

interface hla
hla_layout stlink
hla_device_desc "ST-LINK/V2-1"
hla_vid_pid 0x0483 0x374b


# Optionally specify the serial number of ST-LINK/V2 usb device. ST-LINK/V2
# devices seem to have serial numbers with unreadable characters.
ST-LINK/V2
# firmware version >= V2.J21.S4 recommended to avoid issues with
adapter serial
# number reset issues.
# eg.
#hla_serial "\xaa\xbc\x6e\x06\x50\x75\xff\x55\x17\x42\x19\x3f"


#
# stm32l1 devices support both JTAG and SWD transports.
#

source [find target/swj-dp.tcl]
source [find mem_helper.tcl]


# select swd
transport select hla_swd
#hardware reset
#reset_config none separate
reset_config srst_only srst_nogate connect_assert_srst


if { [info exists CHIPNAME] } {
```

```

    set _CHIPNAME $CHIPNAME
} else {
    set _CHIPNAME stm32l1
}

set _ENDIAN little

# Work-area is a space in RAM used for flash programming
# By default use 10kB
if { [info exists WORKAREASIZE] } {
    set _WORKAREASIZE $WORKAREASIZE
} else {
    set _WORKAREASIZE 0x2800
}

# JTAG speed should be <= F_CPU/6.
# F_CPU after reset is 2MHz, so use F_JTAG max = 333kHz
adapter_khz 240

adapter_nsrst_delay 100
if {[using_jtag]} {
    jtag_nrst_delay 100
}

#jtag scan chain
if { [info exists CPUTAPID] } {
    set _CPUTAPID $CPUTAPID
} else {
    if { [using_jtag] } {
        # See STM Document RM0038
        # Section 30.6.3 - corresponds to Cortex-M3 r2p0
        set _CPUTAPID 0x4ba00477
    } else {
        # SWD IDCODE (single drop, arm)
        set _CPUTAPID 0x2ba01477
    }
}

swj_newdap $_CHIPNAME cpu -irlen 4 -ircapture 0x1 -irmask 0xf -expected-id
$_CPUTAPID

```

```

if {[using_jtag]} {
    jtag newtap $_CHIPNAME bs -irlen 5
}

set _TARGETNAME $_CHIPNAME.cpu
target create $_TARGETNAME cortex_m -endian $_ENDIAN -chain-position
$_TARGETNAME

$_TARGETNAME configure -work-area-phys 0x20000000 -work-area-size
$_WORKAREASIZE -work-area-backup 0

# flash size will be probed
set _FLASHNAME $_CHIPNAME.flash
flash bank $_FLASHNAME stm32lx 0x08000000 0 0 0 $_TARGETNAME

reset_config srst_nogate

if {[using_hla]} {
    # if srst is not fitted use SYSRESETREQ to
    # perform a soft reset
    cortex_m reset_config sysresetreq
}

proc stm32l_enable_HSI {} {
    # Enable HSI as clock source
    echo "STM32L: Enabling HSI"

    # Set HSION in RCC_CR
    mww 0x40023800 0x00000101

    # Set HSI as SYSCLK
    mww 0x40023808 0x00000001

    # Increase JTAG speed
    adapter_khz 1800
}

$_TARGETNAME configure -event reset-init {
    stm32l_enable_HSI
}

```



```

$_TARGETNAME configure -event reset-start {
    adapter_khz 240
}

$_TARGETNAME configure -event examine-end {
    # DBGMCU_CR |= DBG_STANDBY | DBG_STOP | DBG_SLEEP
    mmw 0xE0042004 0x00000007 0

    # Stop watchdog counters during halt
    # DBGMCU_APB1_FZ |= DBG_IWDG_STOP | DBG_WWDG_STOP
    mmw 0xE0042008 0x00001800 0
}

$_TARGETNAME configure -event trace-config {
    # Set TRACE_IOEN; TRACE_MODE is set to async; when using sync
    # change this value accordingly to configure trace pins
    # assignment
    mmw 0xE0042004 0x00000020 0
}

```