UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INGENIERÍA INFORMÁTICA

MÁSTER EN CIBERSEGURIDAD Y SEGURIDAD DE LA INFORMACIÓN

TRABAJO FIN DE MÁSTER

ConPidential: Dispositivo para la custodia de información utilizable como seguro de vida

Mikel Aguirre Zulaika

Septiembre, 2019

UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INGENIERÍA INFORMÁTICA

Departamento de Sistemas Informáticos

TRABAJO FIN DE MÁSTER

ConPidential: Dispositivo para la custodia de información utilizable como seguro de vida

Autor: Mikel Aguirre Zulaika

Director: Pablo González Pérez

Septiembre, 2019

Resumen

En este documento se presenta la memoria de un Trabajo de Fin de Máster realizado en el marco del "Máster en Ciberseguridad y Seguridad de la Información". El objetivo general del trabajo ha sido desarrollar un prototipo de dispositivo que sirva de almacenamiento seguro para información sensible mientras reciba una prueba de vida, dentro de un tiempo determinado; y en caso de no recibirla, envíe la información custodiada a los contactos especificados.

Para el desarrollo y construcción de ese dispositivo, al que hemos llamado "ConPidential", se seleccionaron una serie de tecnologías que permitieran incorporar las funcionalidades deseadas, entre ellas, la autenticación basada en tres factores. En concreto, se optó por basar el proyecto en una Raspberry Pi, a la cual se conectaron diferentes elementos que permitían la interacción del usuario con el dispositivo ConPidential. Dicha plataforma hardware se enlazó con un Bot de Telegram que fue desarrollado para establecer el medio de comunicación para las pruebas de vida. Tras revisar documentación de distintas fuentes y profundizar en cada una de las tecnologías, se diseñó el que sería el funcionamiento de ConPidential. Finalmente, tras integrar todas las piezas y desarrollar las funcionalidades, se obtuvo el resultado final que presentamos, un prototipo con el que se pudieron alcanzar los objetivos establecidos para el TFM.

Esta memoria describe el proceso del trabajo realizado y presenta el resultado final. Además, en las conclusiones, se aportan una serie de reflexiones así como algunas ideas para sofisticar un dispositivo como el construido aquí, o para ampliar sus funcionalidades.

Agradecimientos

A Pablo por su apoyo y por regalarme una gran idea que me ha permitido echarle imaginación y disfrutar del desarrollo de este trabajo.

A Jose Luis por el empeño y el esfuerzo para llevar adelante este Máster.

Dedicatorias

A mi mujer Laura, por aguantarme durante el transcurso del Máster y en especial durante este verano de encierro ;)

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

CAPÍTULO 1. INTRODUCCIÓN

En este primer capítulo introductorio se explican brevemente tanto la motivación que ha servido como punto de inicio y motor para el desarrollo del TFM, como los objetivos de éste. Asimismo, se presenta la estructura de la memoria, para facilitar al lector el seguimiento del resto de documento.

1.1 Motivación

En numerosas ocasiones hemos visto en la ficción (películas, series, comics, etc.) como un agente secreto o espía utilizaba información muy relevante, y casi siempre escandalosa, relacionada con sus posibles captores -o incluso "aliados"-, como seguro de vida. Frases como "si algo me pasa, la información se hará pública" o similares aparecen constantemente en este tipo de historias.

La idea de este proyecto surge precisamente de este recurso utilizado en la ficción por espias, como el conocidísimo agente 007 y otros tantos, pero también referenciado en casos reales tales como Wikileaks. ¿Seríamos capaces de desarrollar un prototipo que pudiera servir para custodiar ese tipo de información altamente sensible como "seguro de vida" con los conocimientos adquiridos durante el transcurso del máster?

De esta idea inicial surge el reto de desarrollar el prototipo "ConPidential", que nos lleva a establecer, para este TFM, los objetivos que a continuación presentamos.

1.2 Objetivos

El objetivo general del presente Trabajo de Fin de Máster ha sido desarrollar un prototipo de dispositivo que sirva de almacenamiento seguro para información sensible mientras reciba una prueba de vida, dentro de un tiempo determinado; y en caso de no recibirla, envíe la información custodiada a los contactos especificados.

Para la consecución de dicho objetivo general, se establecieron una serie de objetivos específicos, muy relacionados con las funcionalidades que el prototipo debía ofrecer:

- Ofrecer al usuario la posibilidad de almacenar información de forma segura y cifrada.
- Contar con mecanismos que faciliten al usuario la interacción con las funcionalidades que éste ofrezca.
- Disponer de un mecanismo de autenticación basado en tres factores (3FA).
- Contar con una vía de comunicación estable con el usuario para la recepción de pruebas de vida por parte del usuario.
- Controlar y verificar la recepción de pruebas de vida y actuar revelando la información almacenada en caso de que no se reciban pruebas de vida válidas antes de que finalice la vigencia de la última prueba de vida.

En el capítulo 8, que recoge las conclusiones del TFM, detallaremos el grado de consecución de estos objetivos inicialmente marcados, y aportaremos una reflexión sobre el alcance del trabajo realizado.

1.3 Estructura de la memoria

Para alcanzar los objetivos especificados en el epígrafe anterior, se ha trabajado a lo largo del tiempo de dedicación al TFM en fases sucesivas. Comenzando por la elección de las tecnologías a utilizar y el diseño del proyecto, y continuando por el análisis de cada una de las “piezas” que finalmente han configurado el prototipo ConPidential, hasta la integración de todas ellas para cumplir con dichos objetivos.

La estructura de esta memoria refleja el trabajo en cada una de estas fases. Tras ofrecer una introducción en este primer capítulo, el segundo capítulo aborda la selección de las tecnologías a emplear en el propotipo, presentando brevemente las características principales de cada una de ellas; además, describe el diseño realizado en esa fase inicial para el desarrollo del prototipo. Seguidamente, se presenta en el tercer capítulo el análisis realizado sobre un módulo de control y sus posibilidades de integración con una Raspberry.

Por su parte, en el capítulo 4 analizamos dos elementos que acturán cómo dispositivos de entrada para nuestro dispositivo. En el quinto capítulo se presenta una de las piezas más relevantes en cuanto a la seguridad se refiere dentro de este proyecto. El sexto capítulo se dedica a presentar la solución elegida para la comunicación de las pruebas de vida por parte del usuario. Y la última fase del trabajo práctico queda reflejada en el capítulo 7, explicando cómo se combinan los distinos elementos y se desarrollan funcionalidades para alcanzar los propósitos que se habían establecido en el TFM.

Finalmente, en el capítulo 8 se presenta una reflexión sobre el alcance del trabajo realizado, y se plantean una serie de propuestas de trabajo futuro, con distintas opciones que se han ido detectando para sofisticar un dispositivo como el construido aquí, o para ampliar sus funcionalidades. La memoria se cierra con la bibliografía, y una descripción de los ficheros que acompañan esta memoria.

Conviene precisar que, siguiendo las recomendaciones sobre citas y referencias bibliográficas en el ámbito de la informática y la electrónica del Servicio de Publicaciones de la UCLM, en esta memoria se emplea el estilo IEEE de citado, a lo largo del texto y en el listado de referencias y enlaces incluido en la bibliografía.

CAPÍTULO 2. SELECCIÓN DE TECNOLOGÍAS Y DISEÑO DEL PROYECTO

En este capítulo se recoge el resultado del trabajo realizado en la primera fase del TFM, una vez establecidos los objetivos. En ese punto, era necesario revisar distintas alternativas, soluciones tecnológicas que, al trabajar de modo conjunto, constituyeran el propotito ConPidential y permitieran alcanzar los objetivos del trabajo.

Tras una fase de búsqueda de información sobre las distintas opciones disponibles, se seleccionaron una serie de tecnologías, cada una de ellas por diversos motivos relacionados con sus funcionalidades y capacidades, su disponibilidad, o capacidad de conexión entre ellas, entre otras razones.

La primera elección estaba relacionada con la plataforma sobre la cual construir el dispositivo ConPidential. La definición inicial de los objetivos principales del proyecto permitía la utilización de básicamente cualquier plataforma hardware. Sin embargo, rápidamente comprobamos que para el cumplimiento de nuevos objetivos que iban surgiendo en esta primera fase de análisis, la solución hardware que mejor encajaba era Raspberry Pi. Las opciones de conectividad tanto contra un mundo puramente IT como contra un mundo cercano al OT (sensores, motores, interruptores, etc.) permitían en esta primera fase temprana del proyecto dar rienda suelta a la imaginación. La potencia de los últimos modelos, la compatibilidad con un mundo de dispositivos, sus dimensiones y su coste fueron claves para su selección.

En un primer boceto del dispositivo, la interacción del usuario con el dispositivo era mínima y se había ideado como una caja negra que actuaba automáticamente tras la conexión de un dispositivo de almacenamiento USB. Durante esta etapa de análisis y diseño surgió la idea de ofrecer un mayor grado de interactividad. Aplicar los conceptos

vistos en la asignatura de IoT y Hardware Hacking resultaba muy interesante y es por ello por lo que decidimos añadir como objetivo la integración de la Raspberry Pi con una placa recuperada de un proyecto basado en Arduino (impresora 3D). Dicha placa nos permitiría elevar considerablemente el grado de interacción entre nuestro dispositivo y el usuario. Además, dado que la placa incluía un lector de tarjetas SD, pensamos en sustituir el medio de entrada de la información a proteger, pasando de un puerto USB que podría ser utilizado como vector de ataque (RubberDucky, BashBunny, etc.) a una solución específica de almacenamiento como es una tarjeta SD.

El siguiente punto en el que trabajamos en esta primera fase fue en las posibilidades de implementación del (3FA) Three Factor Authentication. De los factores "algo que sé", "algo que tengo" y "algo que soy", los dos primeros no tuvieron opción de duda. El primer factor sería implementado mediante un PIN y el segundo factor mediante un token TOTP. En este punto surgió la idea de utilizar otro elemento de un proyecto basado en Arduino, un Keypad que sería utilizado para la introducción de los dígitos del PIN y del TOTP. De cara a cubrir el tercer factor, estudiamos varias opciones existentes. Hoy en día existen multitud de sensores más o menos accesibles que permiten obtener propiedades físicas de un usuario. Por estar implantado cada vez más dispositivos, estudiamos las posibilidades de utilizar el reconocimiento facial, pero el esfuerzo técnico y económico superaba considerablemente el de este proyecto. Tras lo cual se decidió utilizar como tercer factor de autenticación las huellas digitales y por lo tanto la utilización de un lector de huellas.

Centrándonos en el cifrado de la información, surgió la curiosidad de si sería posible integrar en el proyecto una solución hardware orientada al cifrado de la información de forma segura. Pensando en soluciones como podría ser un HSM (Hardware Security Module) encontramos un dispositivo diseñado especialmente para placas Raspberry Pi que por lo menos en base a los documentos técnicos disponibles, cumplía con nuestras expectativas.

La comunicación y el control de las pruebas de vida fue otro de los puntos importantes que nos hizo reflexionar. En un inicio barajamos la solución Latch como medio para el control de las pruebas de vida, pero tras un análisis en profundidad de las posibilidades de su versión community la desechamos por no cumplir con los requisitos fijados. Finalmente, dada su versatilidad, optamos el desarrollo de un bot de Telegram para llevar a cabo dicha función.

Como último punto de esta fase, se estudiaron las posibilidades en cuanto a lenguajes de programación y por facilidad de comprensión, portabilidad y recursos existentes en la red se eligió Pyhton como lenguaje de programación para el desarrollo del proyecto.

2.1 Diseño del proyecto

Tras la definición de los dispositivos y las tecnologías en general que iban a ser utilizadas en el desarrollo del proyecto, se llevó a cabo un diseño del proyecto en cuanto a la definición de las tareas se refiere. A continuación, se muestra la lista de tareas identificadas en esta fase para establecer un calendario y repartir esfuerzos y dedicación para lograr los hitos resultantes de estas tareas:

- Análisis de las opciones de conectividad GPIO de la Raspberry y el resto de los elementos hardware.
- Análisis de la placa Smart Controller y división del estudio y desarrollo de pruebas en ítems más pequeños:
    - Display LCD (Conexionado y desarrollo de pruebas básicas)
    - Rotary Encoder (Conexionado y desarrollo de pruebas básicas)
    - Lector de tarjetas SD (Conexionado, configuración del sistema operativo y desarrollo de pruebas básicas)
- Análisis del lector de huellas, conexionado y desarrollo de pruebas básicas
- Análisis del Keypad, conexionado y desarrollo de pruebas básicas
- Estudio de las posibilidades que ofrece la API de programación sobre Pyhton para el desarrollo de Bots de Telegram y desarrollo de pruebas básicas
- Estudio de las opciones existentes para la revelación de la información almacenada en el dispositivo (FTP, correo, subida a entorno web)
- Utilización de la información recolectada en las fases anteriores para el desarrollo de la solución ConPidential.

CAPÍTULO 3. ANÁLISIS DEL SMART CONTROLLER Y CONEXIÓN A LA RASPBERRY

Tal y como se indicaba en el capítulo anterior, se decidió utilizar el módulo "RepRapDiscount Smart Controller" como medio de interacción del dispositivo ConPidential con el usuario. En este capítulo veremos la circuitería que lo conforma y los pasos dados para la integración de esta placa con la Raspberry.

Como se puede observar en la siguiente figura, en el frontal de la placa nos encontramos con un display LCD, un Buzzer, un Rotary Encoder y un pulsador.

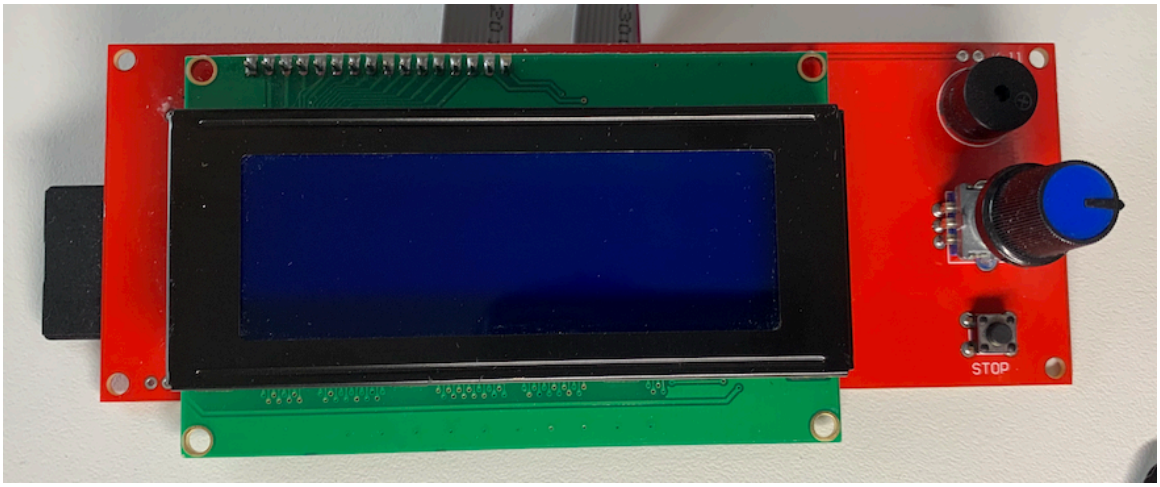

Figura 1. Frontal Smart Controller

Por su parte, en la trasera de la placa, se encuentra el lector de tarjetas SD, el regulador de contraste del LCD, un trasnformador de 5 Voltios a 3.3 Voltios y dos conectores IDC de 10 pines (EXT1 y EXT2).

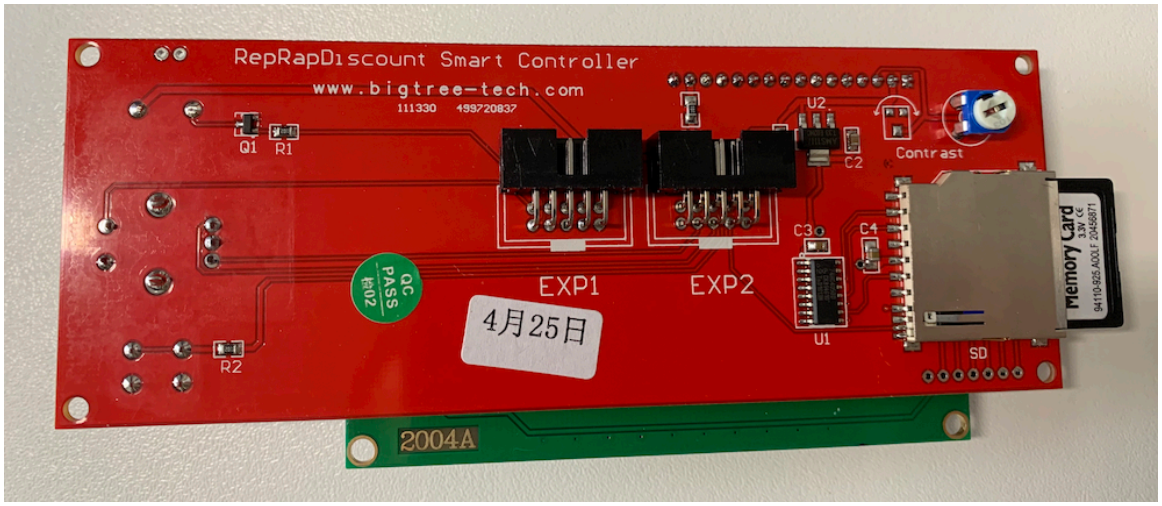

Figura 2. Trasera Smart Controller

3.1 Estudio de la circuitería y conexión con la Raspberry

Tal y como vimos en la asignatura "IoT y Hardware hacking", un análisis visual de un circuito nos puede dar mucha información sobre los elementos que lo conforman y como se encuentran conectados entre ellos. En este caso, no se trataba de una placa realmente compleja y tras un primer vistazo era posible crear un primer esquema con el que ponernos manos a la obra.

Con la ayuda de un multímetro pudimos detectar fácilmente algunas conexiones directas entre pines de los conectores IDC y pines de elementos de la placa, pero nuestro esquema se quedaba realmente corto y es por ello por lo que nos centramos en buscar un diagrama detallado de la placa. Tras una búsqueda exhaustiva, encontramos dos esquemas muy detallados de la placa que nos sirvieron de gran ayuda para el desarrollo de esta fase del proyecto.

Una vez aclarado el esquema de conexiones de la placa, era necesario comprender cómo funcionaban cada uno de los elementos que la conformaban y las opciones de interconexión que ofrecía la Raspberry por medio de sus puertos GPIO.

3.1.1 Display LCD

Tras el análisis inicial, el primer elemento con el que nos pusimos a trabajar fue el display LCD. Se trata de un display 2004A retroiluminado que cuenta con 4 líneas y 20 columnas. El controlador HD44780 instalado en el LCD nos permite representar en cada

una de esas celdas uno de los caracteres que tiene preprogramados, además de los 8 patrones extra que nos permite customizar para la utilización de símbolos y caracteres personalizados [1].

El primer paso consistió en leer documentación variada sobre cómo funcionaba el controlador HD44780 y cuales eran las posibles formas de conectar un display LCD como este a una Raspberry utilizando los puertos GPIO. Un artículo de la página web Circuit Basics [2] nos sirvió de gran ayuda para la conexión del display a nuestra Raspberry y la posterior realización de las primeras pruebas de funcionamiento.

En dicho artículo se mostraban dos formas de conectar la placa 2004A a una Raspberry:

- 8 Bit Mode (Utilizando los 8 Pines de Datos)
- 4 Bit Mode (Utilizando los 4 últimos Pines de Datos)

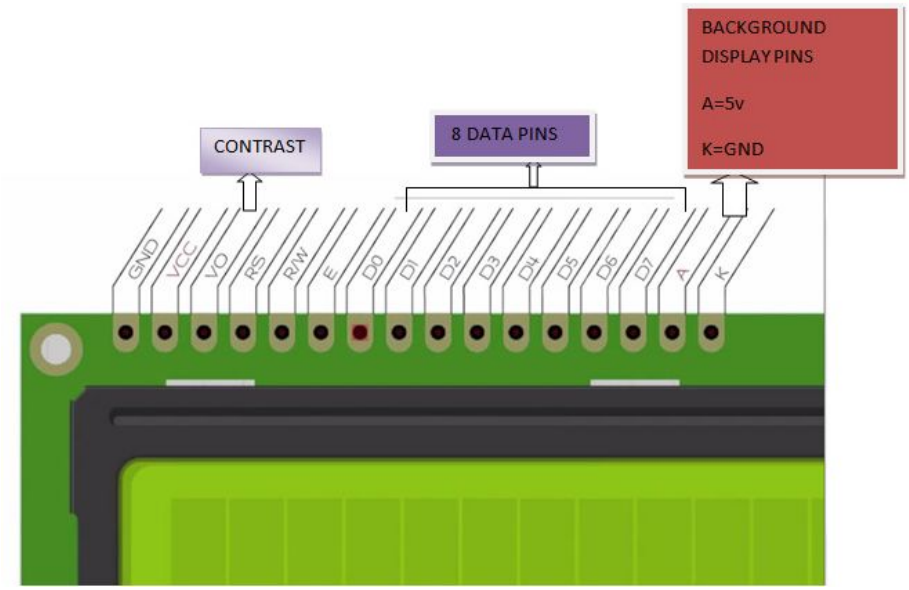

Figura 3. Pines LCD 2004A

Dado que la placa Smart Controller tenía todos los pines del LCD soldados, era necesario revisar los esquemas correspondientes a la circuitería de la placa para poder comprobar si dichos pines llegaban a los conectores IDC de forma directa (sin pasar por ningún controlador) y en caso de llegar directos, cuántos de ellos quedaban accesibles.

Como se puede observar en la siguiente figura, sólo los últimos 4 pines de datos del módulo LCD (LCD4, LCD5, LCD6 y LCD7) se encontraban realmente conectados y eran accesibles directamente junto con los pines LCDRS y LCDE a través del conector IDC EXP1.

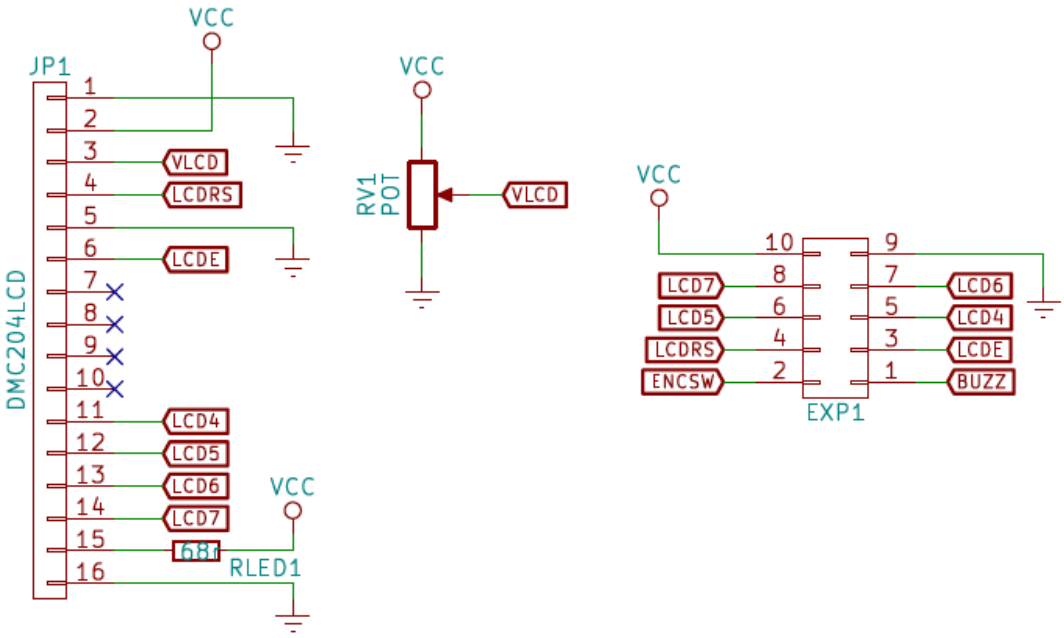

Figura 4. Conexiones LCD Smart Controller

Teniendo en cuenta que la placa ya contaba con un potenciómetro para el control del contraste, el esquema de conexión entre la Raspberry y la placa Smart Controller sería muy similar a este:

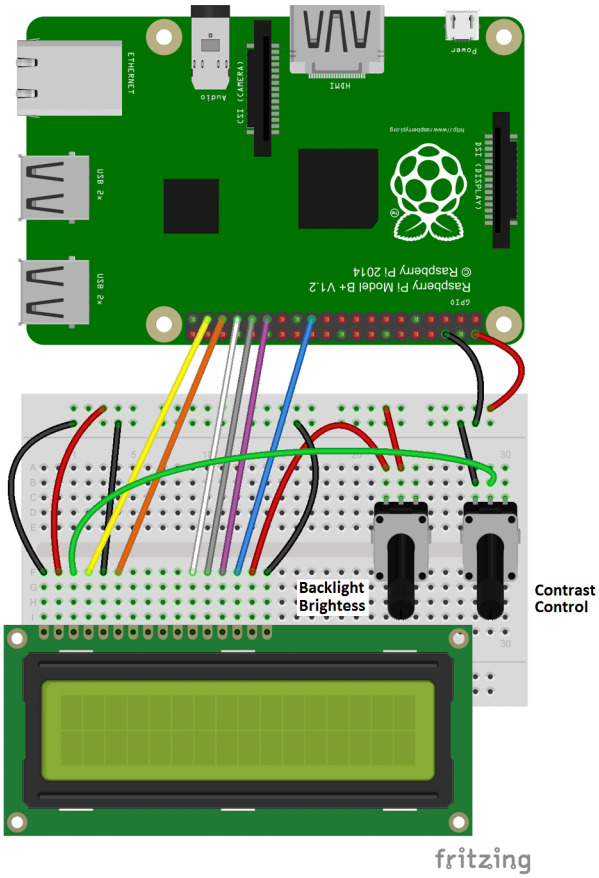

Figura 5. Conexión LCD a Raspberry Pi

Debíamos por tanto seleccionar 6 puertos GPIO de nuestra Raspberry para establecer la conexión de control (2 pines) y datos (4 pines) contra el módulo LCD, y además debíamos conectar las líneas VCC (5 voltios) y GND para alimentar el módulo.

SEÑAL	PIN CONECTOR IDC	PIN GPIO RPi	COLOR CABLE
LCDRS	EXP1 - 4	GPIO 17 – PIN 11	ROJO
LCDE	EXP1 - 3	GPIO 18 – PIN 12	MARRON
LCD4	EXP1 - 5	GPIO 27 – PIN 13	NEGRO
LCD5	EXP1 - 6	GPIO 22 – PIN 15	BLANCO
LCD6	EXP1 - 7	GPIO 23 – PIN 16	GRIS
LCD7	EXP1 - 8	GPIO 24 – PIN 18	MORADO

Tabla 1. Esquema conexiones módulo LCD

Tras establecer las conexiones indicadas y conectar el Pin 10 del conector EXT1 a 5V y el Pin 9 a GND, comprobamos que el LCD era funcional. Sólo nos quedaba comprobar si éramos capaces de enviar texto a nuestro display por medio de un pequeño script programado en Python. Basándonos en el script ***lcd2004.py*** que encontramos en la página http://domoticx.com [3] durante la búsqueda de información, logramos llevar a cabo una sencilla prueba en la que escribíamos texto en pantalla ocupando las 4 líneas y 20 columnas disponibles. Habíamos conseguido por tanto el propósito de esta fase.

3.1.2 Rotary Encoder

El siguiente paso consistía en analizar el funcionamiento del Rotary Encoder, establecer la conexión con la Raspberry y desarollar un pequeño script que reaccionase a los movimientos y pulsaciones.

Tras un primer análisis de la información encontrada al respecto en varias webs, el funcionamiento del Rotary Encoder se presentaba relativamente simple y todo indicaba que no íbamos a tener grandes problemas para su control desde un script basado en Python.

Esquemas como el que podemos ver en la siguiente figura, nos daba una idea clara de cual era el mecanismo por el cual el controlador, en este caso la Raspberry, sería capaz de detectar si estábamos realizando un movimiento en sentido de las agujas del reloj o en el contrario. Los dos puntos de la figura se corresponden con las líneas **CLK** y **DT** cuyo valor variará según estén "libres" o entren en contacto con la rueda dentada que en la figura representa a la pieza principal del Rotary Encoder.

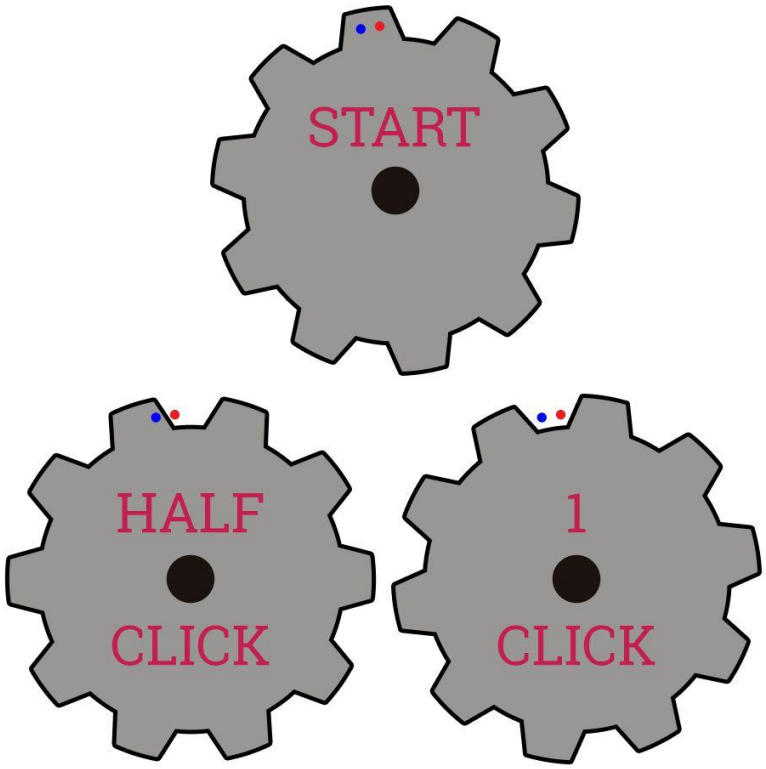

Figura 6. Esquema1 funcionamiento Rotary Encoder

Toda la documentación encontrada en relación con módulos Rotary Encoder para proyectos Maker tenían algo en común: todos contaban con conexión VCC. Por tanto, cuando la rueda hacía contacto con una de las dos líneas (CLK y DT), éstas pasaban de GND a estar conectadas a VCC, o lo que es lo mismo del valor 0 al valor 1. Sin embargo, como vemos a continuación, el Rotary Encoder de nuestro Smart Controller no cuenta con conexión VCC y por lo tanto cuando se cierra el contacto con las líneas CLK y DT, estas pasan a estar conectadas a GND.

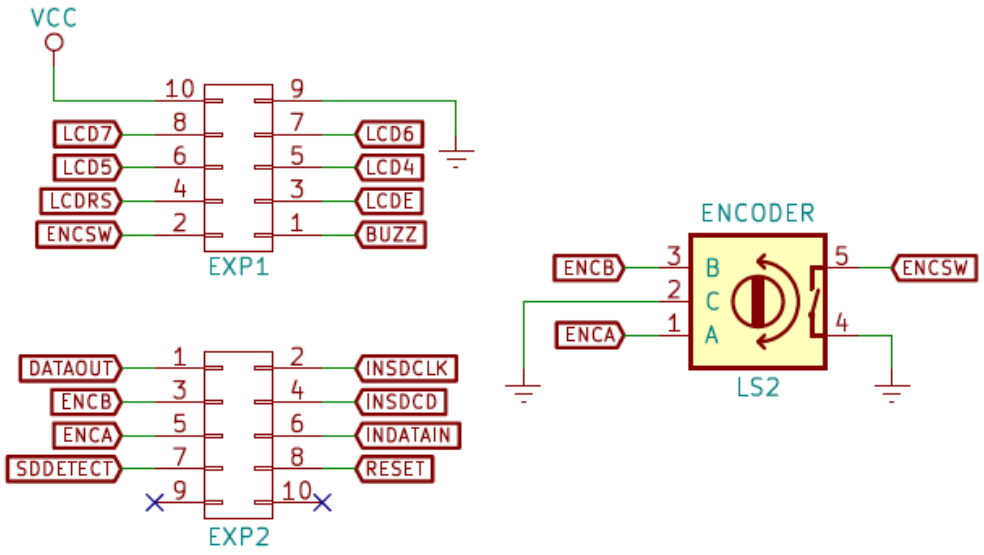

Figura 7. Esquema conexiones Rotary Encoder Smart Controller

Este cambio respecto a placas muy extendidas como puede ser la **KY040** tiene dos implicaciones importantes. En primer lugar, dado que la conexión se realiza contra GND, la posición en la que las líneas CLK y DT (ENCA y ENCB en el esquema anterior) se encuentran desconectadas debe quedar a VCC en el lado de la Raspberry. Para ello, la Raspberry incorpora una serie the **Pull-Up Resistors** que podemos configurar por software para que la línea GPIO correspondiente se encuentre conectada a VCC hasta que el circuito conectado a ella derive a GND.

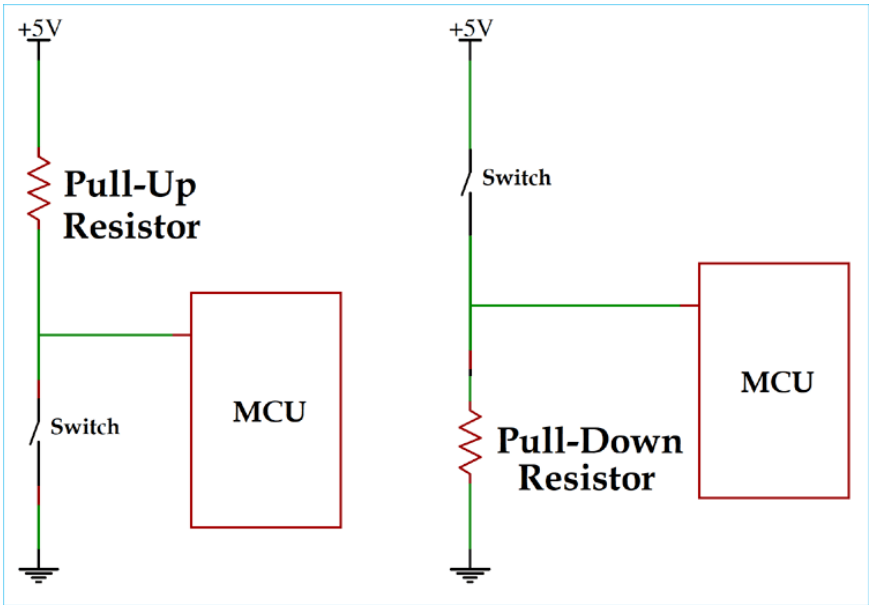

Figura 8. Pull-Up y Pull-Down Resistors

En segundo lugar, los estados por los que pasan las líneas CLK y DT en este modelo ligado a GND no son los mismos que en el caso de la placa **KY040,** y por lo tanto las librerías escritas en Python y los ejemplos encontrados durante el análisis de la información no nos sirven para nuestro circuito.

SEÑAL	PIN CONECTOR IDC	PIN GPIO RPi	COLOR CABLE
CLK	EXP2 - PIN5	GPIO 20 – PIN 38	ROJO
DT	EXP2 - PIN3	GPIO 21 – PIN 40	MARRON
SW	EXP1 - PIN2	GPIO 7 – PIN 26	VIOLETA

Tabla 2. Esquema conexiones Rotary Encoder

Tras conectar los pines correspondientes al Rotary Encoder, tal y como se indica en la tabla anterior, realizamos una serie de pruebas modificando código Python que sumaba o restaba valores según girábamos el Encoder hacia la derecha o hacia la izquierda pero fuimos incapaces de conseguir un correcto funcionamiento hasta que realizamos la siguiente tabla:

	Movimiento Derecha		Movimiento Izquierda	
	CLK	**DT**	**CLK**	**DT**
Estado Inicial	**1**	**1**	**1**	**1**
Medio Click	**0**	**1**	**1**	**0**
1 Click	**0**	**0**	**0**	**0**
Medio Click	**1**	**0**	**0**	**1**
1 Click	**1**	**1**	**1**	**1**

Tabla 3. Esquema movimiento Rotary Encoder

Si tomamos como referencia la línea CLK, de cara a saber si estamos moviéndonos hacia la derecha o hacia la izquierda nos interesa controlar las posiciones de la tabla seleccionadas en verde. Se trata de los puntos en los que el valor de CLK cambia y en los que, según sea el valor de DT, podremos detectar si el movimiento es en el sentido de las agujas del reloj o en el contrario.

Tras plasmar este esquema en código Python, conseguimos controlar correctamente el movimiento del Rotary Encoder y realizar sumas y restas según el sentido del movimiento.

No fue este el único punto peliagudo con el Rotary Encoder, el control de la pulsación nos dio algún que otro quebradero de cabeza por detectarnos pulsaciones dobles. Tras varias pruebas, finalmente conseguimos eliminar las pulsaciones fantasma utilizando el concepto de **debounce time** (permitir una sola interrupción cada X ms y desechar el resto).

3.1.3 Lector de tarjetas SD

El tercer y último elemento del Smart Controller con el que íbamos a trabajar era el lector de tarjetas SD incorporado en la parte trasera de la placa. Así, comenzamos esta fase revisando información sobre el funcionamiento de este tipo de tarjetas en relación con los pines, las señales y las posibilidades de conexión con equipos/controladores. Artículos como “How to Use MMC/SDC” [4] nos sirvieron para tener una idea clara de cuáles eran las bases del funcionamiento de las tarjetas SD y comprender las funcionalidades de cada una de las líneas identificadas en el Smart Controller y conectadas al lector de tarjetas.

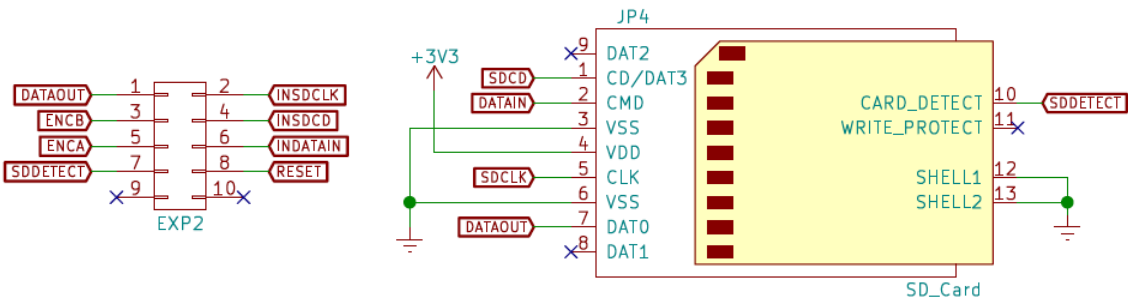

Figura 9. Esquema conexiones lector tarjetas SD

Tras el análisis de la circuitería correspondiente al lector de tarjetas SD del módulo Smart Contoller, se realizó una búsqueda de información acerca de las posibilidades de configuración del sistema operativo de la Raspberry (Raspbian) para la conexión de un segundo lector de tarjetas SD. En el estudio se encontró un artículo llamado "*Adding a secondary sd card on Raspberry PI*" [5] en el que se detallaban dos formas de añadir un segundo lector de tarjetas SD a la Raspberry:

- Habilitando el segundo controlador SDIO
- Utilizando el controlador SPI

Tal y como se indica en el artículo, trabajar con el segundo controlador SDIO tiene como ventaja la velocidad que se alcanza frente a la alcanzada mediante el controlador SPI; pero como desventaja, limita a uno el número de lectores que podemos conectar. En nuestro caso, dado que los pines GPIO indicados para la conexión mediante el segundo controlador SDIO han sido utilizados para la conexión con el display LCD, y dado que el lector sólo se iba a utilizar en modo lectura y para el volcado de datos al dispositivo, optamos por la opción del controlador SPI.

Para lo cual, llevamos a cabo el siguiente conexionado entre la placa Smart Controller y los puertos GPIO de la Raspberry:

Name	SD Card	Raspberry Pi - GPIO
VCC	4	17 - 3.3v
GND	6	20 - GND
CLK/SCLK	5	23 - SPI0 CLK
CMD/MOSI	2	19 - SPI0 MOSI
DAT0/MISO	7	21 - SPI0 MISO
DAT1	8	
DAT2	9	
DAT3/CS	1	24/0 - SPI0 CE0 N

Tabla 4. Conexionado lector SD - SPI

Tras el conexionado y su correspondiente verificación, el siguiente paso fue crear un fichero “Device Tree Source (DTS)” [6] con el siguiente contenido:

```
/dts-v1/;
/plugin/;

/ {
   compatible = "brcm,bcm2835", "brcm,bcm2836", "brcm,bcm2708",
"brcm,bcm2709";

   fragment@0 {
      target = <&spi0>;
      frag0: __overlay__ {
         status = "okay";
         sd1 {
                reg = <0>;
                status = "okay";
                compatible = "spi,mmc_spi";
                voltage-ranges = <3000 3500>;
                spi-max-frequency = <8000000>;
         };
      };
   };
};
```

Tras la instalación del paquete device-tree-compiler (*sudo apt install device-tree-compiler*), se procede a la compilación del fichero DTS con el que generamos el fichero DTBO que será cargado por el Kernel en el aranque:

```
mikeljuice@raspberrypi:~/Desktop/TFM/SD $ sudo dtc -@ -I dts -O dtb -o mmc_spi.dtbo mmc_spi.dts
mmc_spi.dtbo: Warning (reg_format): /fragment@0/__overlay__/sd1:reg: property has invalid length (4 bytes) (#address-cells == 2, #size-cells == 1)
mmc_spi.dtbo: Warning (unit_address_vs_reg): /fragment@0/__overlay__/sd1: node has a reg or ranges property, but no unit name
mmc_spi.dtbo: Warning (pci_device_reg): Failed prerequisite 'reg_format'
mmc_spi.dtbo: Warning (pci_device_bus_num): Failed prerequisite 'reg_format'
mmc_spi.dtbo: Warning (simple_bus_reg): Failed prerequisite 'reg_format'
mmc_spi.dtbo: Warning (avoid_default_addr_size): /fragment@0/__overlay__/sd1: Relying on default #address-cells value
mmc_spi.dtbo: Warning (avoid_default_addr_size): /fragment@0/__overlay__/sd1: Relying on default #size-cells value
mmc_spi.dtbo: Warning (avoid_unnecessary_addr_size): Failed prerequisite 'avoid_default_addr_size'
mmc_spi.dtbo: Warning (unique_unit_address): Failed prerequisite 'avoid_default_addr_size'
```

Figura 10. Compilado Device Tree Source (DTS)

Para que el Kernel pueda cargar el DTBO creado es necesario copiar el fichero resultante de la compilación en la ruta “***/boot/overlays/***” y posteriormente modificar el fichero “***/boot/config.txt***” añadiendo la referencia al módulo del Kernel **mmc_spi**:

```
 GNU nano 3.2                                                    /boot/config.txt

#config_hdmi_boost=4

# uncomment for composite PAL
#sdtv_mode=2

#uncomment to overclock the arm. 700 MHz is the default.
#arm_freq=800

# Uncomment some or all of these to enable the optional hardware interfaces
#dtparam=i2c_arm=on
#dtparam=i2s=on
#dtparam=spi=on

# Uncomment this to enable the lirc-rpi module
#dtoverlay=lirc-rpi

# Additional overlays and parameters are documented /boot/overlays/README

# Enable audio (loads snd_bcm2835)
dtparam=audio=on

[pi4]
# Enable DRM VC4 V3D driver on top of the dispmanx display stack
dtoverlay=vc4-fkms-v3d
max_framebuffers=2

[all]
#dtoverlay=vc4-fkms-v3d

# Lector SD por SPI
dtoverlay=mmc_spi
```

Figura 11. Fichero /boot/config.txt

Tras el reinicio del SO, revisamos el fichero ***syslog*** y el volcado del commando ***dmesg*** para comprobar si el driver del Kernel había sido correctamente cargado y el lector de tarjetas SD había sido reconodido.

```
raspberrypi kernel: [  611.971007] mmc_spi spi0.0: /aliases ID not available
raspberrypi kernel: [  614.999166] mmc_spi spi0.0: SD/MMC host mmc3, no WP, no poweroff, cd polling
raspberrypi kernel: [  615.008754] mmc_spi spi0.0: no support for card's volts
raspberrypi kernel: [  615.008771] mmc3: error -22 whilst initialising SDIO card
raspberrypi kernel: [  615.013685] mmc_spi spi0.0: no support for card's volts
raspberrypi kernel: [  615.013703] mmc3: error -22 whilst initialising SD card
raspberrypi kernel: [  615.015596] mmc_spi spi0.0: no support for card's volts
raspberrypi kernel: [  615.015609] mmc3: error -22 whilst initialising MMC card
raspberrypi kernel: [  619.112648] mmc_spi spi0.0: no support for card's volts
raspberrypi kernel: [  619.112655] mmc3: error -22 whilst initialising SDIO card
```

Figura 12. Referencias al driver mmc_spi en syslog

Como podemos ver en la figura anterior, el driver fue cargado correctamente por parte del Kernel y se reconoció el dispositivo mmc3. Sin embargo, comprobamos que el log del sistema reportaba de continuo las siguientes dos líneas:

- mmc_spi spi0.0: no support for card's volts

- mmc3: error -22 whilst initialising MMC card

Una búsqueda de información en Internet nos llevó a un artículo relacionado con la configuración de mmc_spi para la placa de desarrollo VoCore [7], en el que se indicaba que el SoC del VoCore no permitía la modificación de los rangos de voltaje para la SD. La solución al problema pasaba por eliminar la línea *"voltage-ranges = <3000 3500>;"* del fichero DTS creado anteriormente y realizar de nuevo el compilado para obtener un nuevo fichero DTBO.

Tras la compilación del fichero DTS modificado, reiniciamos el SO y revisamos de nuevo el contenido del fichero syslog en busca de referencias al módulo mmc_spi.

```
raspberrypi kernel: [    4.902975] mmc_spi spi0.0: /aliases ID not available
raspberrypi kernel: [    7.934317] mmc_spi spi0.0: SD/MMC host mmc3, no WP, no poweroff, cd polling
raspberrypi kernel: [    7.937964] mmc_spi spi0.0: no support for card's volts
raspberrypi kernel: [    7.941024] mmc_spi spi0.0: no support for card's volts
raspberrypi kernel: [    7.941982] mmc_spi spi0.0: no support for card's volts
raspberrypi kernel: [   11.987851] mmc_spi spi0.0: no support for card's volts
raspberrypi kernel: [   11.990860] mmc_spi spi0.0: no support for card's volts
raspberrypi kernel: [   11.991916] mmc_spi spi0.0: no support for card's volts
raspberrypi kernel: [   16.067842] mmc_spi spi0.0: no support for card's volts
raspberrypi kernel: [   16.070826] mmc_spi spi0.0: no support for card's volts
raspberrypi kernel: [   16.071744] mmc_spi spi0.0: no support for card's volts
raspberrypi kernel: [   20.147823] mmc_spi spi0.0: no support for card's volts
raspberrypi kernel: [   20.150808] mmc_spi spi0.0: no support for card's volts
raspberrypi kernel: [   20.153543] mmc_spi spi0.0: no support for card's volts
raspberrypi kernel: [  105.834548] mmc_spi spi0.0: no support for card's volts
raspberrypi kernel: [    4.848180] mmc_spi spi0.0: /aliases ID not available
raspberrypi kernel: [    4.848207] mmc_spi spi0.0: OF: voltage-ranges unspecified
raspberrypi kernel: [    4.848219] mmc_spi spi0.0: ASSUMING 3.2-3.4 V slot power
raspberrypi kernel: [    7.874345] mmc_spi spi0.0: SD/MMC host mmc3, no WP, no poweroff
raspberrypi kernel: [    7.877950] mmc_spi spi0.0: no support for card's volts
raspberrypi kernel: [    7.881007] mmc_spi spi0.0: no support for card's volts
raspberrypi kernel: [    7.884665] mmc_spi spi0.0: no support for card's volts
```

Figura 13. Syslog tras recompilación fichero mmc_spi.dts

Como podemos observar en la figura anterior, el Kernel nos lanza un warning indicando que no se han especificado "voltage-ranges" y nos indica que se asume el rango 3.2 – 3.4 Voltios. Después de este warning, el Kernel se silencia y no reporta de continuo el error "***no support for card's volts***" que vemos al inicio de la figura. Sin embargo, nos percatamos de que, tras la modificación, la opción "***cd polling***" desaparece de la línea en la que el Kernel reconoce y carga el dispositivo. Esto provoca que, tal y como indicamos en el capítulo anterior, dado que no hemos podido activar el "SD Detect" del lector, el Kernel realice una única prueba de detección de la tarjeta SD durante el arranque y no vuelva a tratar de detectar la existencia de ésta.

Tras lo cual, se revisaron multitud de artículos y entradas de foros especializados en el desarrollo de drivers para el Kernel de Linux [8] y tras no llegar a ninguna conclusión clara, se reescribió el código del fichero DTS con el fin de evitar los warnings de compilación que veíamos en la figura 4.

```
dts-v1/;
/plugin/;

/ {
   compatible = "brcm,bcm2835", "brcm,bcm2836", "brcm,bcm2708",
"brcm,bcm2709";

   fragment@0 {
      target = <&spi0>;
      __overlay__ {
         #address-cells = <1>;
         #size-cells = <0>;
         status = "okay";
         sd1@0 {
                reg = <0>;
                status = "okay";
                compatible = "spi,mmc_spi";
                voltage-ranges = <3000 3500>;
                spi-max-frequency = <8000000>;
         };
      };
   };
};
```

```
mikeljuice@raspberrypi:~/Desktop/TFM/SD $ sudo dtc -@ -I dts -O dtb -o mmc_spi.dtbo mmc_spi.dts
mikeljuice@raspberrypi:~/Desktop/TFM/SD $ sudo cp mmc_spi.dtbo /boot/overlays/
```

Figura 14. Compilación DTS versión 2

Tras la compilación y el reinició, comprobamos como al insertar una tarjeta SD en el lector, ésta era reconocida como el dispositivo **/dev/mmcblk3**, con su única partición referenciada por medio de **/dev/mmcblk3p1**.

```
[    7.794334] mmc_spi spi0.0: SD/MMC host mmc3, no WP, no poweroff, cd polling
[    8.156357] mmc3: host does not support reading read-only switch, assuming write-enable
[    8.156369] mmc3: new SDHC card on SPI
[    8.157046] mmcblk3: mmc3:0000 APPSD 7.68 GiB
[    8.163427]  mmcblk3: p1
```

Figura 15. Detección tarjeta SD - Dispositivo mmcblk3

El siguiente paso era configurar el sistema para que, tras detectar la inserción de una tarjeta SD en el lector, fuese capaz de montar automáticamente el sistema de ficheros existente en la tarjeta en una ruta determinada. Para ello, se pensó en utilizar el sistema UDEV y preparar una regla que llevase a cabo el montaje de la partición primaria de la tarjeta SD tras la detección de su inserción en el lector de tarjetas. Tras varias pruebas, se

comprobó que las reglas creadas sobre UDEV no funcionaban correctamente porque UDEV no está pensado para lanzar comandos que requieran tiempo de ejecución. Finalmente, encontramos una alternativa basada en UDEV más SystemD más un script bash [9].

- Script Bash: **/usr/local/bin/sd_spi-mount.sh:**

```
#!/bin/bash

usage()
{
    echo "Usage: $0 {add|remove} device_name (e.g. sdb1)"
    exit 1
}

if [[ $# -ne 2 ]]; then
    usage
fi

ACTION=$1
DEVBASE=$2
DEVICE="/dev/${DEVBASE}"

# See if this drive is already mounted, and if so where
MOUNT_POINT=$(/bin/mount | /bin/grep ${DEVICE} | /usr/bin/awk '{ print $3 }')

do_mount()
{
    if [[ -n ${MOUNT_POINT} ]]; then
        echo "Warning: ${DEVICE} is already mounted at ${MOUNT_POINT}"
        exit 1
    fi

    # Get info for this drive: $ID_FS_LABEL, $ID_FS_UUID, and $ID_FS_TYPE
    eval $(/sbin/blkid -o udev ${DEVICE})

    # Figure out a mount point to use
    #LABEL=${ID_FS_LABEL}
    # Modified by Mikel to allways use the same label
    LABEL="SD_EXT"
    if [[ -z "${LABEL}" ]]; then
        LABEL=${DEVBASE}
    elif /bin/grep -q " /media/${LABEL} " /etc/mtab; then
        # Already in use, make a unique one
        LABEL+="-${DEVBASE}"
    fi
    MOUNT_POINT="/media/${LABEL}"

    echo "Mount point: ${MOUNT_POINT}"

    /bin/mkdir -p ${MOUNT_POINT}

    # Global mount options
    OPTS="rw,relatime"
```

```
# File system type specific mount options
    if [[ ${ID_FS_TYPE} == "vfat" ]]; then
        OPTS+=",users,gid=100,umask=000,shortname=mixed,utf8=1,flush"
    fi

    if ! /bin/mount -o ${OPTS} ${DEVICE} ${MOUNT_POINT}; then
        echo "Error mounting ${DEVICE} (status = $?)"
        /bin/rmdir ${MOUNT_POINT}
        exit 1
    fi

    echo "**** Mounted ${DEVICE} at ${MOUNT_POINT} ****"
}

do_unmount()
{
    if [[ -z ${MOUNT_POINT} ]]; then
        echo "Warning: ${DEVICE} is not mounted"
    else
        /bin/umount -l ${DEVICE}
        echo "**** Unmounted ${DEVICE}"
    fi

    # Delete all empty dirs in /media that aren't being used as mount
    # points. This is kind of overkill, but if the drive was unmounted
    # prior to removal we no longer know its mount point, and we don't
    # want to leave it orphaned...
    for f in /media/* ; do
        if [[ -n $(/usr/bin/find "$f" -maxdepth 0 -type d -empty) ]]; then
            if ! /bin/grep -q " $f " /etc/mtab; then
                echo "**** Removing mount point $f"
                /bin/rmdir "$f"
            fi
        fi
    done
}

case "${ACTION}" in
    add)
        do_mount
        ;;
    remove)
        do_unmount
        ;;
    *)
        usage
        ;;
esac
```

El script **sd_spi-mount.sh** está basado en el script **usb-mount.sh** disponible en la página referenciada y realiza la función de montar o desmontar un determinado dispositivo de almacenamiento. El script original estaba ideado para el montaje del dispositivo en una ruta dinámica generada en base al nombre de dispositivo. Dado que en nuestro caso nos interesaba que la tarjeta SD siempre se montase en la misma ruta, modificamos la línea resaltada para especificar como nombrede dispositivo **SD_EXT**.

Tal y como hemos visto antes, el sistema referenciaba la tarjeta SD como el dispositivo de bloque /dev/mmcblk3. Por lo tanto, necesitábamos que UDEV reaccionase a eventos "add" y "remove" correspondientes a dicho dispositivo. Para ello, generamos el siguiente fichero de reglas que llamaría al sistema SystemD para pedir el arranque o parada de un determinado servicio. Esta llamada no requeriría a UDEV quedarse a la espera de la ejecución y respuesta de otros comandos, y por tanto superaríamos el hándicap que nos habíamos encontrado.

- Reglas UDEV: **/etc/udev/rules.d/99-local.rules**

```
KERNEL=="mmcblk3p[0-9]", ACTION=="add", RUN+="/bin/systemctl start sd_spi-mount@%k.service"

KERNEL=="mmcblk3p[0-9]", ACTION=="remove", RUN+="/bin/systemctl stop sd_spi-mount@%k.service"
```

El servicio **sd_spi-mount@%k.service** sería el nexo de unión entre el script **/usr/local/bin/sd_spi-mount.sh** y la regla UDEV**.**

- Fichero **/etc/systemd/system/sd_spi-mount@.service**

```
[Unit]
Description=Montar/Desmontar SD externa en %i
[Service]
Type=oneshot
RemainAfterExit=true
ExecStart=/usr/local/bin/sd_spi-mount.sh add %i
ExecStop=/usr/local/bin/sd_spi-mount.sh remove %i
```

Dado que, como comentábamos anteriormente, no conseguimos integrar la señal SD-DETECT en nuestro dispositivo, la única forma de trasladar al sistema un evento de extracción de la tarjeta SD del lector sería desactivando el módulo del Kernel mmc_spi. Mediante la ejecución de la instrucción "***sudo modprobe -r mmc_spi***" forzamos la descarga del módulo correspondiente y por lo tanto UDEV debería detectar un evento "**remove**" del dispositivo mmcblk3.

```
raspberrypi kernel: [ 6685.667124] mmc3: SPI card removed
raspberrypi systemd[1]: Stopping Montar/Desmontar SD externa en mmcblk3p1...
raspberrypi systemd[485]: media-SD_EXT.mount: Succeeded.
raspberrypi systemd[862]: media-SD_EXT.mount: Succeeded.
raspberrypi systemd[1]: media-SD_EXT.mount: Succeeded.
raspberrypi sd_spi-mount.sh[2014]: **** Unmounted /dev/mmcblk3p1
raspberrypi kernel: [ 6685.823371] FAT-fs (mmcblk3p1): unable to read boot sector to mark fs as dirty
raspberrypi sd_spi-mount.sh[2014]: **** Removing mount point /media/SD_EXT
raspberrypi systemd[1]: sd_spi-mount@mmcblk3p1.service: Succeeded.
raspberrypi systemd[1]: Stopped Montar/Desmontar SD externa en mmcblk3p1.
```

Figura 16. Desmotado SD_EXT tras evento Remove

Tras activar de nuevo el módulo mmc_spi mediante la instrucción "***sudo modprobe mmc_spi***", el sistema queda de nuevo a la espera de un evento "**Add**" por parte de UDEV. Al detectar la inserción de la tarjeta SD, se lanza el servicio **sd_spi-mount** y la partición primaria de la tarjeta SD queda montada en la ruta **/media/SD_EXT.**

```
raspberrypi kernel: [ 5788.178238] mmc_spi spi0.0: /aliases ID not available
raspberrypi kernel: [ 5791.208330] mmc_spi spi0.0: SD/MMC host mmc3, no WP, no poweroff, cd polling
raspberrypi kernel: [ 5791.636833] mmc3: host does not support reading read-only switch, assuming write-enable
raspberrypi kernel: [ 5791.636845] mmc3: new SDHC card on SPI
raspberrypi kernel: [ 5791.637629] mmcblk3: mmc3:0000       7.55 GiB
raspberrypi kernel: [ 5791.643878]  mmcblk3: p1
raspberrypi systemd[1]: Starting Montar/Desmontar SD externa en mmcblk3p1...
raspberrypi sd_spi-mount.sh[1864]: Mount point: /media/SD_EXT
raspberrypi sd_spi-mount.sh[1864]: **** Mounted /dev/mmcblk3p1 at /media/SD_EXT ****
raspberrypi systemd[1]: Started Montar/Desmontar SD externa en mmcblk3p1.
```

Figura 17. Montado SD_EXT tras evento Add

De este modo, el Smart Controller quedaba conectado a la Rapsberry y configurado para la fase final de integración con el resto de elementos del proyecto.

CAPÍTULO 4. KEYPAD Y LECTOR DE HUELLAS

Como indicábamos en el capítulo 2, de cara a la entrada de datos relacionada con la autenticación del usuario contra el dispositivo, definimos dos elementos hardware: un keypad y un lector de huellas digitales. A continuación, abordamos las tareas que se llevaron a cabo durante la fase de diseño del proyecto para hacer posible la integración de estos dos elementos en nuestro dispositivo ConPidential.

4.1 Keypad

Figura 18. Keypad 3x4

Seguramente se trate del elemento más sencillo de este proyecto y el que menos quebraderos de cabeza nos originó.

Utilizamos como base un keypad de membrana de 4 filas x 3 columnas que forman una matriz formada por 12 interruptores conectados cada uno de ellos a dos líneas, una correspondiente a la fila en la que se encuentran y otra correspondiente a la columna. De esta forma, el keypad entrega 7 líneas de las cuales 2 quedarán conectadas entre sí tras la pulsación de uno de los interruptores. Relacionando la fila y la columna que queden conectadas, obtendremos la posición del elemento dentro de la matriz.

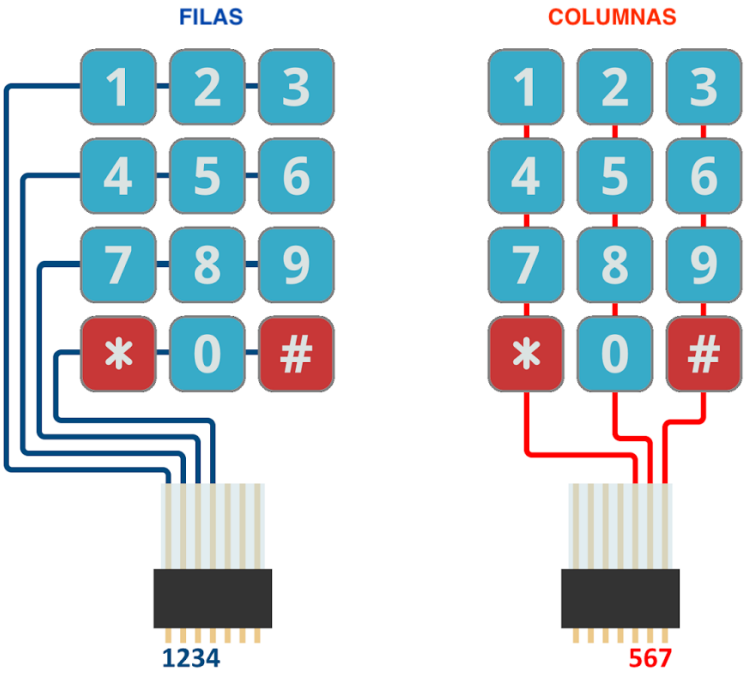

Figura 19. Esquema circuito Keypad

La conexión de este elemento a la Raspberry no tenía dificultad alguna, ya que no requería la utilización de un tipo de puerto determinado ni era necesario proveerlo de alimentación. Como inconveniente a tener en cuenta, el número de conexiones GPIO necesarias para el despliegue del keypad en nuestro proyecto era elevado. Al tratarse de una conexión paralelo se consumen un número elevado de puertos GPIO que podrían ser utilizados para soportar conexiones más interesantes y complejas.

LINEA	PIN GPIO RPi	COLOR CABLE
1	GPIO 26 – PIN 37	AZUL
2	GPIO 16 – PIN 36	VERDE
3	GPIO 19 – PIN 35	AMARILLO
4	GPIO 5 – PIN 29	NARANJA
5	GPIO 6 – PIN 31	ROJO
6	GPIO 12 – PIN 32	MARRON
7	GPIO 13 – PIN 33	GRIS

Tabla 5. Esquema conexiones Keypad

Pero no todo eran desventajas y la facilidad de despliegue del Keypad nos permitió dedicar tiempo a analizar un proyecto llamado CircuitPython [10] que como veremos más adelante utilizamos finalmente en el desarrollo de la solución software. Se trata de un proyecto elaborado por Adafruit que comenzó como una variante de MicroPython (versión reducida de Python pensada para poder "caber" en microcontroladores), añadiendo a ésta la capacidad de trabajar con LEDs, sensores, motores, etc.

El proyecto creció y llevó la API desarrollada en un inicio para microcontroladores a entornos Linux desplegados sobre placas base con pines GPIO disponibles, como es la

Raspberry. La ventaja de esta API es la portabilidad ya que es ésta la que se integra con las librerías del sistema que dan acceso al hardware (RPi.GPIO en el caso de la Raspberry).

Utilizando CircuitPython y la librería *adafruit_matrixkeypad*, desarrollamos un script que más adelante evolucionaríamos y utilizaríamos para la gestión del PIN del usuario. Dicho script nos pedía introducir un PIN utilizando el Keypad y al pulsar el sexto dígito, la lista de dígitos pulsados se convertía en un string y se calculaba el SHA256 de dicho string para compararlo con un SHA256 almacenado. Si ambos HASH coincidían, el PIN había sido correctamente introducido.

```
mikeljuice@raspberrypi:~/Desktop/TFM/keypad-rpi $ sudo python3 test.py
Introduzca su PIN:
1
5
9
7
5
3
[1, 5, 9, 7, 5, 3]
159753
3d14c2d4e4ced81e459e4ace7c01466a700000fb94a3bbe944a55fb92693e879
PIN correcto
```

Figura 20. Script comprobación PIN – Keypad

La idea de evitar almacenar ningún PIN en claro -ni en el código ni en un fichero de la solución- era una idea relacionada con la seguridad presente desde el inicio del proyecto. Como veremos más adelante, esta aproximación se mantuvo y se optó por la utilización de SHA256 para el almacenamiento y comprobación del PIN de autenticación del usuario.

4.2 Lector de huellas

Tras haber decidido en la fase de análisis del proyecto que íbamos a basar el control biométrico en la huella digital y haber seleccionado un dispositivo lector de huellas concreto, comprobamos que teníamos una pequeña limitación. Dado el número de pines GPIO que habíamos consumido hasta el momento y la necesidad de reservar los 10 primeros para el módulo Zymkey, y teniendo en cuenta que el lector de huellas nos entregaba una conexión UART, decidimos adquirir y utilizar un conversor USB-TTL CP2102 para llevar a cabo la conexión de nuestro lector con la Raspberry.

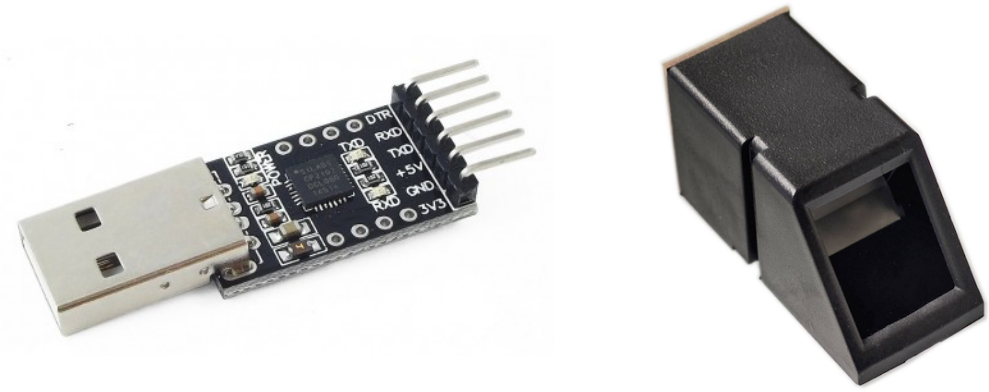

Figura 21. Lector de huellas y conversor USB-TTL

Tras leer y analizar el manual de usuario del lector JM-101 y conectarlo a la Raspberry por medio del conversor USB, realizamos una búsqueda de información relacionada con posibles librerías y/o API existentes para Python que nos permitiesen interactuar de manera "sencilla" con el lector de huellas. La búsqueda tuvo su fruto y encontramos un artículo muy interesante en el que integraban un lector similar con una Raspberry, utilizando también un conversor USB-TTL, y lo más importante, utilizaban una serie de librerías para Python que facilitaban de manera muy considerable el trabajo con el lector [11].

Tal y como indicaban en el artículo, tras añadir al sistema de paquetes APT el repositorio de "PM Code Works", instalamos el paquete ***python-fingerprint*** el cual, además de las librerías necesarias, incluía una serie de scripts de ejemplo que permitían realizar tareas de gestión básicas sobre el lector. Tras comprobar que el puerto serie facilitado por el conversor USB se encontraba accesible en nuestro sistema (*/dev/ttyUSBXX*), lanzamos la ejecución de las siguientes pruebas y comprobamos su correcto funcionamiento:

- Registro de una nueva huella en la memoria del lector:

```
mikeljuice@raspberrypi:~ $ python3 /usr/share/doc/python3-fingerprint/examples/example_enroll.py
Currently used templates: 0/300
Waiting for finger...
Remove finger...
Waiting for same finger again...
Finger enrolled successfully!
New template position #0
```

Figura 22. Ejemplo1 script lector huellas

- Revisión de los registros almacenados en la memoria del lector:

```
mikeljuice@raspberrypi:~ $ python3 /usr/share/doc/python3-fingerprint/examples/example_index.py
Currently used templates: 1/300
Please enter the index page (0, 1, 2, 3) you want to see: 0
Template at position #0 is used: True
Template at position #1 is used: False
Template at position #2 is used: False
Template at position #3 is used: False
Template at position #4 is used: False
```

Figura 23. Ejemplo2 script lector huellas

- Lectura de una huella y comprobación de su existencia en la memoria del lector:

```
mikeljuice@raspberrypi:~ $ python3 /usr/share/doc/python3-fingerprint/examples/example_search.py
Currently used templates: 1/300
Waiting for finger...
Found template at position #0
The accuracy score is: 109
SHA-2 hash of template: 8c59d9a9f634832a17590e0875be63669135475985bef7df53fd9bbc80e84dd9
```

Figura 24. Ejemplo3 script lector huellas

Con ello, el keypad y lector de huellas quedaban listos para su uso posterior, cuando los integráramos con el resto de las tecnologías en el prototipo ConPidential.

CAPÍTULO 5. MODULO ZYMKEY

En este capítulo se presenta una de las piezas clave del proyecto. Como indicábamos al inicio, una de las preocupaciones respecto a la seguridad del dispositivo ConPidential era la gestión del cifrado de la información almacenada en éste. Dada la criticidad de la información almacenada en el dispositivo -no olvidemos que se trata de información utilizable como seguro de vida-, dicha información no debería ser fácilmente accesible por un tercero. Teniendo en cuenta que nuestro prototipo estaba basado en una Raspberry Pi, cuyo almacenamiento principal se encuentra en una tarjeta microSD, era de vital importancia asegurar que en caso de que el dispositivo cayese en malas manos, la información no pudiese ser extraida de una forma tan simple como montar la tarjeta microSD en otro equipo para acceder a la información.

Partiendo de esta premisa, la primera opción que se analizó fue el cifrado completo del sistema de ficheros desplegado en la microSD. Dado que íbamos a trabajar sobre un SO basado en Linux, la utilización de LUKS (Linux Unified Key Setup) y dm-crypt era la opción lógica. Sin embargo, tras analizar más en profundidad las necesidades del dispositivo encontramos un punto a tener muy en cuenta: la gestión y el almacenamiento de la Keyfile. Una vez cifrado el sistema de ficheros correspondiente a la raíz, para que el Kernel pudiese acceder a la información deberíamos proveer al mismo de una passphrase. Esta passphrase podría ser introducida por el usuario en el arranque (no viable por el tipo de dispositivo) o se encontraría volcada en un fichero (Keyfile) que a su vez debería estar almacenada en un sistema de ficheros accesible por el Kernel, convirtiendo a este punto en crítico.

De cara a ofrecer un almacenamiento realmente seguro de la información y de las claves de encriptación se pensó en la posibilidad de utilizar un módulo HSM (Hardware Security Module):

"Un HSM es un dispositivo criptográfico basado en hardware que genera, almacena y protege claves criptográficas y suele aportar aceleración hardware para operaciones criptográficas." [12]

Un análisis de soluciones HSM existentes en el mercado para soluciones IoT y más concretamente para soluciones basadas en Raspberry Pi, nos llevó a dar con el dispositivo Zymkey.

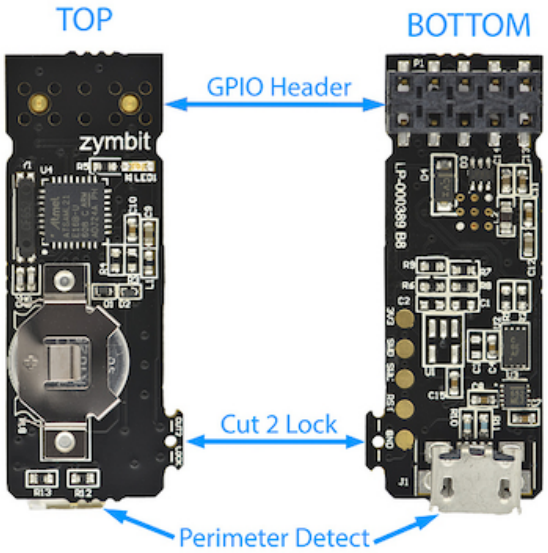

Figura 25. Módulo Hardware Zymkey

Zymkey 4i es un módulo de seguridad I2C desarrollado por la empresa Zymbit para proveer a la Raspberry Pi de las siguientes funcionalidades:

- Generación de un token único en base a una serie de propiedades específicas de la placa base sobre la que se despliegue
- Motor criptográfico basado en algoritmos tales como ECDSA, ECDH, AES-256, SHA256
- Keystore basado en hardware para el almacenamiento seguro de pares de claves pública/privada
- RTC – Real Time Clock
- Monitorización en busca de síntomas de manipulación física

Todas estas propiedades convertían al módulo Zymkey en un elemento ideal para nuestro dispositivo. Si volvemos atrás y recuperamos el problema de la gestión y el almacenamiento de la Keyfile necesaria para el acceso por parte de LUKS a la Master Key del volumen cifrado, esta solución se integra con LUKS y permite cifrar y descifrar un sistema de ficheros sin tener que almacenar la Keyfile en plano. Según la documentación de la solución, la integración con LUKS permite el cifrado y firmado de la Keyfile por parte del módulo Zymkey de forma que, si el módulo Zymkey no se encontrase operativo o si se hubiese detectado un intento de manipulación, el sistema de ficheros de la microSD quedaría inaccesible.

La secuencia de arranque de la Raspberry quedaría de la siguiente forma:

1. El Kernel inicializa el sistema initramfs
2. Initramfs entrega la Keyfile cifrada y firmada al módulo Zymkey
3. Zymkey valida la firma y si es correcta, descifra la Keyfile
4. La Keyfile descifrada se entrega a LUKS y el sistema de ficheros root queda descifrado

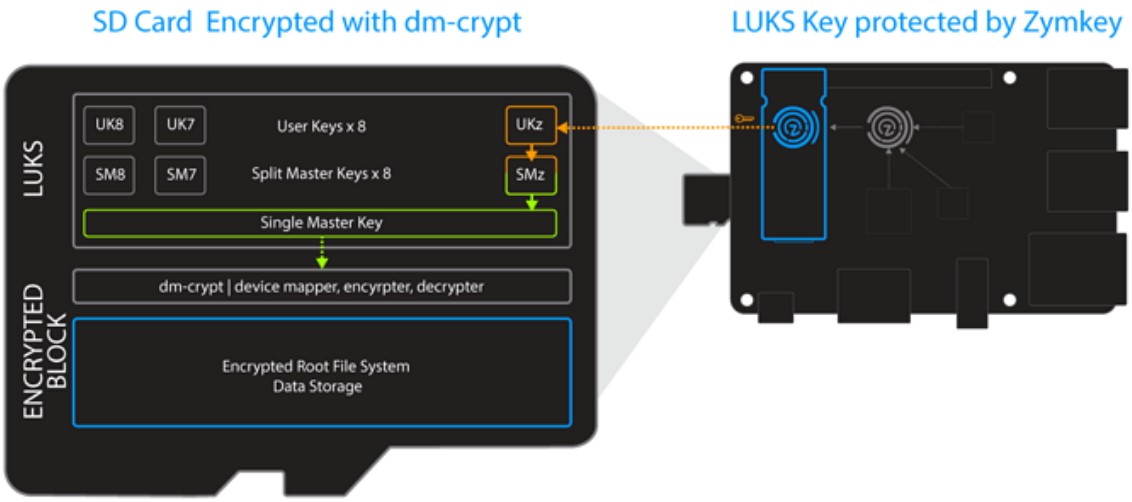

Figura 26. Integración Zymkey + LUKS

La integración con LUKS es una funcionalidad muy importante, pero no es la única que sería beneficiosa para nuestro dispositivo. Poder cifrar la información crítica que el usuario desearía almacenar, independientemente del cifrado del sistema de ficheros y descargando a la Raspberry de la gestión y almacenamiento de las claves de cifrado, eleva el nivel de seguridad de nuestra solución de forma considerable.

Como indicábamos en las funcionalidades del módulo, Zymkey genera un token único que identifica el dispositivo unívocamente (Device ID). Dicho token se genera utilizando datos tales como el número de serie de la placa base y el de la tarjeta microSD y, combinando este token con la clave simétrica única que se encuentra almacenada en el módulo, se crea un enlace (binding) que es utilizado para la autenticación del acceso al módulo Zymkey. Una vez es generado este enlace entre el módulo y el resto de elementos del sistema, en caso de suplantación de alguno de ellos, el enlace quedaría roto y los mecanismos de autenticación de Zymkey evitarían cualquier acceso al módulo. De esta manera, un intento de suplantación de la tarjeta microSD dejaría inservible nuestro dispositivo ConPidential.

El módulo cuenta con dos modos de funcionamiento: Modo desarrollador y Modo Producción. En el primero, que ha sido el utilizado durante el desarrollo del proyecto, el módulo Zymkey queda ligado temporalmente al bundle Raspberry + microSD y es posible resetear este enlace en cualquier momento para proceder a una reinstalación o desarrollo de

un nuevo proyecto. El modo Producción, en cambio, una vez configurado es irreversible y el funcionamiento del módulo Zymkey queda enlazado al dispositivo de por vida. Esto significa que ante una modificación en el sistema (suplantación de tarjeta microSD o sustitución de placa Raspberry) el sistema de autenticación del módulo Zymkey fallaría y no permitiría el acceso a las claves almacenadas en el módulo, haciendo imposible el acceso a cualquier dato cifrado con dichas claves.

Si relacionamos esta funcionalidad con la monitorización en busca de síntomas de manipulación física, el alcance aumenta considerablemente. El dispositivo cuenta con un giroscopio y un acelerómetro integrados que permiten detectar movimientos bruscos y/o golpeos que podrían indicar intentos de manipulación. Tras la detección sería posible actuar en consecuencia como, por ejemplo, lanzando una notificación (a través de API). Pero la detección de manipulaciones puede ser elevada un paso más mediante el uso del módulo Protokit desarrollado por la misma empresa. Dicho módulo está formado por dos shields que se instalan sobre la Raspberry y se conectan con el módulo Zymkey por medio del puerto USB llamado “Perimeter detect”. En caso de que el dispositivo montado sobre el Protokit se tratara de desmontar, el módulo Zymkey detectaría una manipulación en el circuito creado por los shields y podría proceder al borrado de la información del binding dejando inservible el módulo Zymkey.

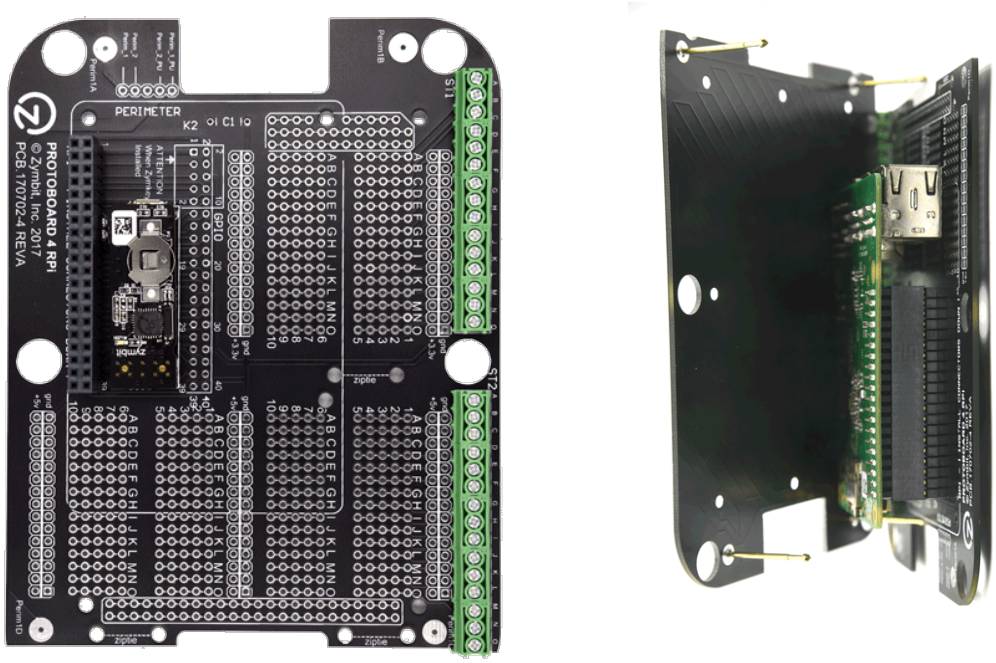

Figura 27. Protokit + Zymkey

La última funcionalidad que destacar, pero no por ello menos importante para este proyecto, es la de contar con un RTC (Real Time Clock) protegido por batería. Teniendo en cuenta que una de las funciones más importantes del dispositivo es la de controlar la recepción de pruebas de vida por parte del usuario detro de periodos de tiempo determinados, es necesario contar con un reloj de precisión que no dependa de sincronización con fuentes externas (NTP) y no sea sensible a cargas de trabajo del procesador principal u otras

interferencias. Es por ello que identificamos esta funcionalidad como interesante para el proyecto y la utilizamos en el desarrollo, como veremos más adelante.

Dado que Zymkey contaba con una API Python, encajaba perfectamente en nuestro proyecto y pudimos sacar partido de las distintas funcionalidades aquí mencionadas.

En cuanto a la conexión del módulo Zymkey con la Raspberry, el módulo está diseñado para ser conectado contra los 10 primeros pines GPIO de la Raspberry. Entre estos pines se encuentran líneas de alimentación de 3.3 y 5 voltios, GND y las líneas que dan soporte al protocolo I2C. Dado que anteriormente habíamos reservado las conexiones necesarias, no nos encontramos con problemas a la hora de realizar el conexionado y comprobar el funcionamiento del módulo.

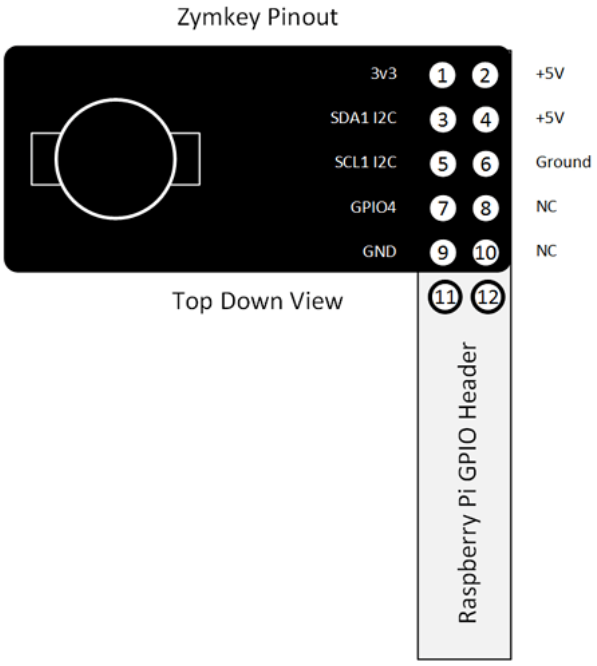

Figura 28. Conexión GPIO módulo Zymkey

La instalación del software necesario para la integración del módulo Zymkey con la Raspberry Pi y el proceso de binding (enlazado) del módulo con el dispositivo fue totalmente transparente. Tras contactar con soporte por estar fuera de línea el servidor desde el que se servían los paquetes de instalación, el servicio fue restablecido y el proceso de instalación finalizó sin error alguno. Una vez instalado el software y activado el enlace, pudimos comprobar cómo entre los scripts de gestión del módulo existía uno que permitía llevar a cabo la migración de una instalación base del SO Raspbian en el que no se encontraba LUKS habilitado a una situación final en la que el sistema de ficheros root se encontraba cifrado y gestionado en el arranque por el sistema LUKS en conjunto con el módulo Zymkey. Todo ello mediante un proceso guiado y con la única necesidad de un dispositivo de almacenamiento USB utilizado para almacenar temporalmente un tarball con el contenido de la tarjeta microSD durante la generación del volumen LUKS.

Con ello, el módulo Zymkey quedaba preparado para la integración con el resto de tecnologías en la fase final del trabajo.

CAPÍTULO 6. @KITT_PIBOT

Si en el capítulo anterior hablábamos de uno de los elementos más significativos del proyecto, en este sexto capítulo vamos a hablar del otro: el bot de Telegram. Tal y como comentábamos al inicio, tras un primer análisis de las opciones a utilizar para la gestión de la comunicación de la prueba de vida por parte del usuario, optamos por el desarrollo de un bot de Telegram.

En un segundo análisis más en profundidad, definimos que dicho bot debería interacturar únicamente con los usuarios previamente definidos en la configuración del dispositivo ConPidential, quedaría a la espera de recepción de pruebas de vida que deberían incluir un elemento de autenticación dinámico y sería el encargado de controlar las ventanas de tiempo entre notificaciones. De modo que, en caso de no recepción de pruebas de vida dentro de la ventana de tiempo definida, éste fuera el encargado de lanzar una serie de acciones.

Partiendo de dicha idea, el primer paso antes de comenzar con el proceso de creación del bot era darle un nombre. Puestos a hacer un homenaje a un "bot" de nuestra infancia, que mejor bot que:

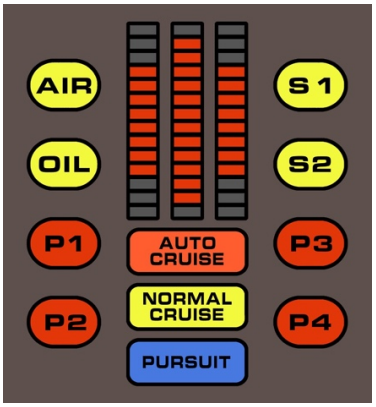

Figura 29. KITT

Una vez definido el nombre, el siguiente paso era ponernos en contacto con el "padrino" de los bot: **@botfather**

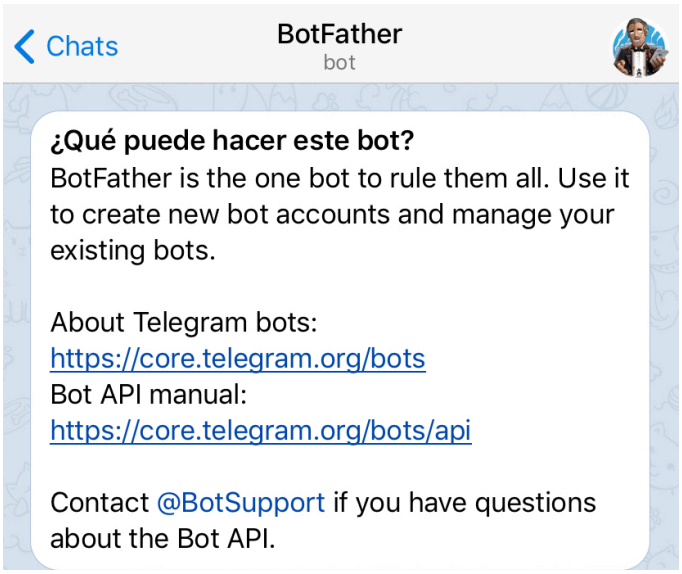

Figura 30. Hello BotFather

Como podemos ver en la figura anterior, BotFather es el bot que se ocupa de gestionar las altas y modificaciones de bots en Telegram. Una vez iniciada la conversación con BotFather, este nos muestra la lista de comandos que podemos utilizar en nuestra interacción.

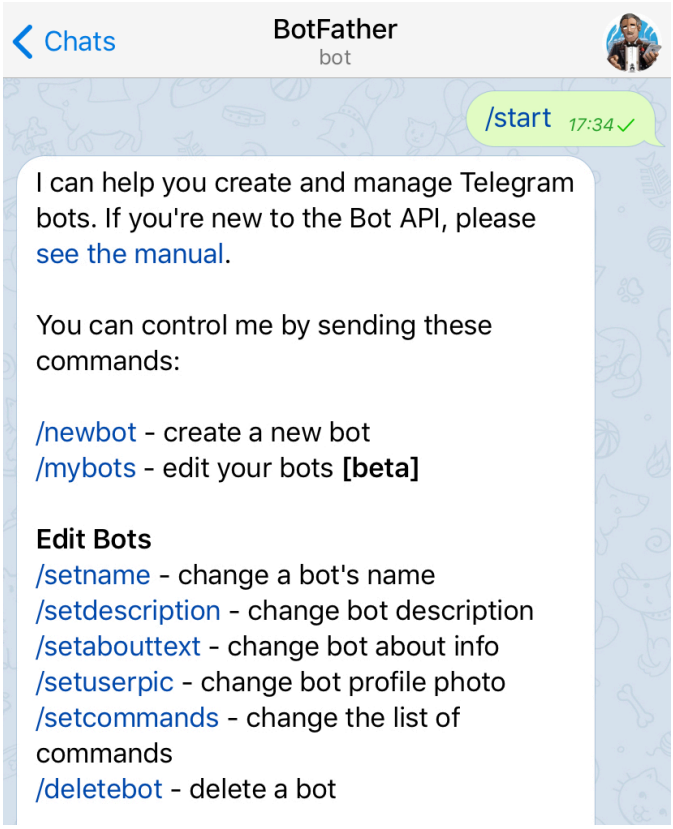

Figura 31. BotFather Commands

Mediante el uso del comando **/newbot**, creamos el bot KITT_Pibot y tras el procesamiento de la petición, BotFather nos respondió indicándonos que KITT_Pibot había sido correctamente generado y nos facilitaba el token correspondiente para el acceso a la API.

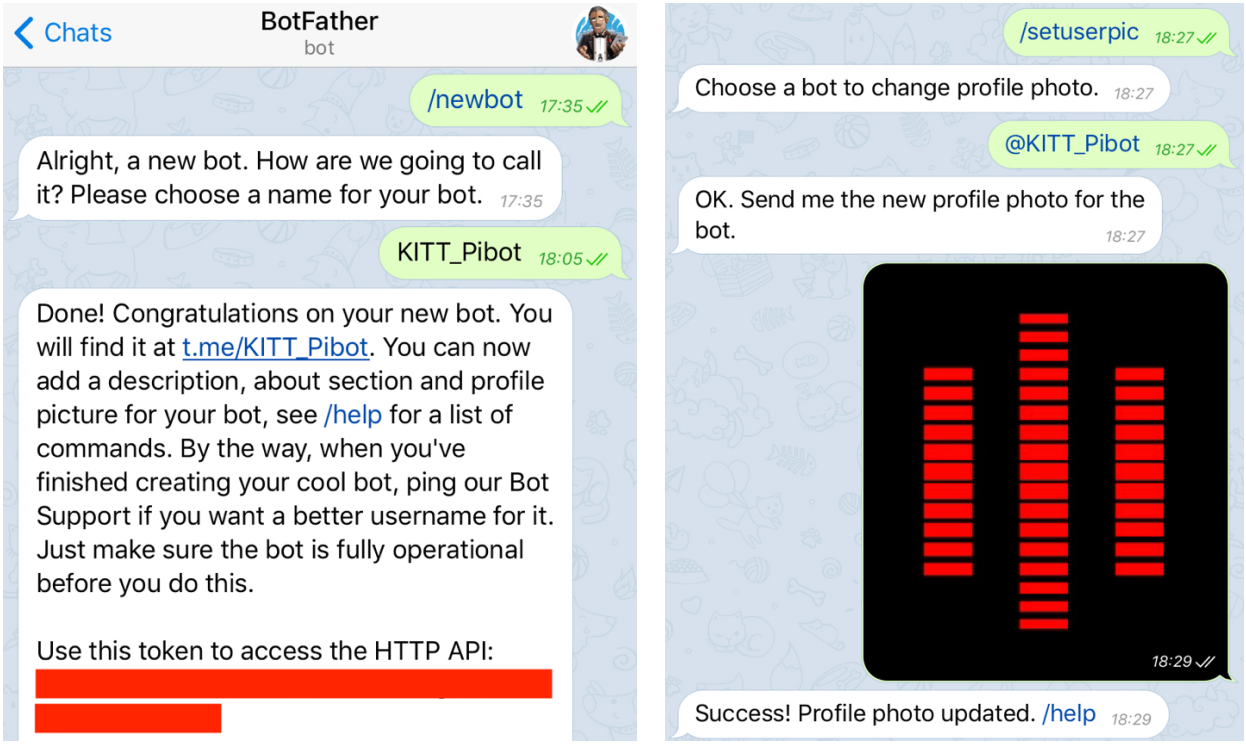

Figura 32. Comandos /newbot y /setuserpic

De cara a personalizar el look de nuestro boot, utilizamos el comando **/setuserpic** para enviarle la imagen que el bot presentaría como foto de perfil y mediante el uso del comando **/setjoingroups**, deshabilitamos la posibilidad de que nuestro bot fuese añadido a un grupo (como medida de seguridad/"privacidad").

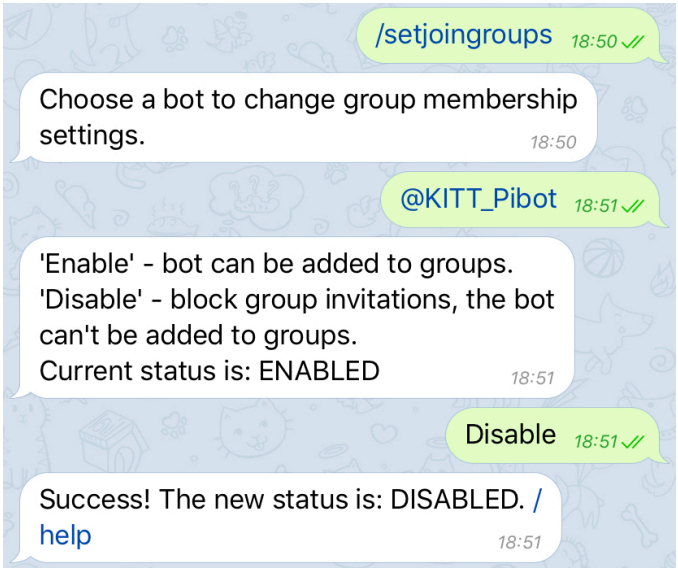

Figura 33. Comando /setjoingroups

El siguiente paso sería la definición de los comandos disponibles para los usuarios. Se trata de una propiedad informativa que facilita el acceso a los distintos comandos que el bot publica (no esta ligado con el desarrollo del bot y de los comandos implementados realmente). Mediante el comando **/mybots** accederíamos a la lista de bots creados desde nuestra cuenta. Si tuviésemos varios Bots, seleccionaríamos el bot KITT_Pibot y BotFather nos mostraría un menú en el que, tras seleccionar la opción "**Bot Settings**" seguida de "**Edit Commands**", se nos indicaría cómo enviar la lista de comandos disponibles (un comando por línea seguido de su descripción).

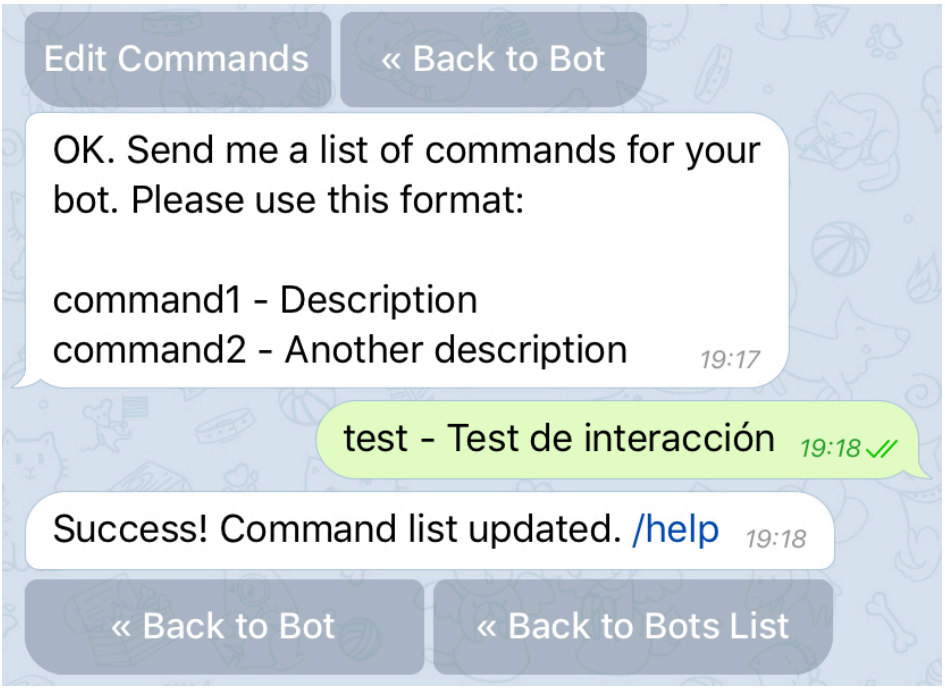

Figura 34. Bot command list

Llegados a este punto, nuestro KITT_Pibot se encontraba correctamente dado de alta en la infraestructura de Telegram. El siguiente paso consistió en analizar en profundidad las funcionalidades que nos ofrecía la API Python de Telegram de cara a desarrollar una pequeña prueba que nos permitiera interactuar con el bot mediante los comandos **/start** (comando de inicio de conversación) y **/test**.

Tras una búsqueda de información, la lectura de varios artículos y la revisión de la Wiki disponible en el repositorio Github de Python-telegram-bot [13] teníamos una idea clara de las necesidades y de la estructura de código que debíamos escribir para nuestra prueba de concepto.

El primer paso consistía en instalar las librerías correspondientes a la API en nuestro entorno Python:

```
sudo pip3 install python-telegram-bot
```

Tras la instalación de las librerías, nos centramos en 4 referencias que habíamos apuntado de la página **Code snippets** dentro de la Wiki antes indicada:

- Post a text message [14]
- Post a gif from a URL (send_animation) [15]
- Restrict access to a handler (decorator) [16]
    - Permitir únicamente a una lista de usuarios el acceso a las funciones que gestionan los comandos del bot
- Send action while handling command (decorator) [17]
    - Enviar "actions" al cliente Telegram del usuario mientras se llevan a cabo los procesos ligados al comando enviado por el usuario. Por ejemplo, enviar el action "escribiendo…" mientras preparamos la respuesta al comando recibido.

Utilizando estas cuatro plantillas de código Python pudimos desarrollar un primer programa muy sencillo que lanzaba un **Updater** en modo **polling** y manejaba un **Dispatcher** para la recogida de comandos y envío de respuestas. Desarrollamos una función llamada **quien_te_envia()** que recogía el ID del usuario que había establecido la comunicación para obtener los IDs de usuario que registraríamos en la lista de usuarios autorizados y desarrollamos funciones **Handler** para los comandos **/start** y **/test** que hacían referencia a los decorators "**Restrict access to a handler**" y "**Send action while handling command**". De esta forma, tras añadir los IDs de usuarios a la lista de autorizados, los comandos /start y /test sólo serían accesibles por estos usuarios y desde que se recibiese el comando hasta que se finalizase el proceso correspondiente, aparecería en el cliente del usuario el estado "Escribiendo…".

El comando **/start** obtendría el nombre de usuario de quien hubiese lanzado dicho comando, mandaría un mensaje de bienvenida y posteriormente un GIF animado. El comando /test comenzaría en un inicio únicamente enviando un mensaje de texto de respuesta y sería utilizado para realizar pruebas de funcionalidades que nos interesara analizar para comprobar si encajaban y eran útiles para nuestro desarrollo.

Figura 35. Test primer desarrollo KITT_Pibot

Comprendido el funcionamiento de un bot de Telegram y de los elementos de la API utilizados para poder desarrollar un bot sencillo, estábamos preparados para integrar las funcionalidades necesarias para los casos de uso analizados y diseñados en el proyecto.

CAPÍTULO 7. INTEGRACIÓN DE LAS TECNOLOGÍAS Y DESARROLLO DE LA SOLUCIÓN

Tras el análisis de las piezas más importantes del puzzle, llegamos a la fase en la que las unimos y nos centramos en el desarrollo de la solución. En la fase inicial del proyecto marcamos una serie de objetivos basados en funcionalidades y en este capítulo vamos a revisar los pasos llevados a cabo tanto en la parte de diseño como en el desarrollo para satisfacer dichos objetivos, empleando el conocimiento adquirido en las fases anteriores.

Figura 36. Prototipo Dispositivo ConPidential

7.1 Interacción con el usuario: Display LCD

Comenzaremos con uno de los objetivos que surgieron nada más tener clara la idea inicial que daba forma al proyecto ConPidential: contar con una solución que facilitara la interactividad del usuario con el dispositivo ConPidential. Como comentamos al inicio, este objetivo tuvo gran peso a la hora de optar por la integración de la Smart Controller en el dispositivo. La utilización de una pantalla LCD y el Rotary Encoder nos permitirían proveer de esa interactividad al usuario sin la necesidad de contar con un teclado y monitor o de una conexión remota a un entorno Shell o Web. Además, se trataría de una solución que reduciría las posibilidades de explotación de vulnerabilidades y por lo tanto de accesos al sistema más allá de la aplicación. Durante las fases anteriores, la idea de interactividad había ido evolucionando hasta el punto en el que nos planteábamos desarrollar un menú multinivel que nos permitiera sacar partido al display y facilitar las modificaciones y el escalado del proyecto.

Tras la revisión de varios ejemplos de desarrollo en Python de menús en modo terminal (consola), realizamos una serie de pruebas de implementación de la misma idea llevándola a un display LCD. Las pruebas realizadas eran demasiado estáticas y no nos permitirían contar con esa facilidad para la introducción de modificaciones y del escalado en general. Fue entonces cuando decidimos realizar una búsqueda en GitHub de proyectos en Python relacionados con menús y displays LCD. Tras revisar varios repositorios con soluciones que no encajaban con nuestras necesidades, dimos con un proyecto que cumplía con nuestros objetivos, con el único obstáculo de que estaba desarrollada para trabajar con displays de 16x2 en vez de 20x4: *https://github.com/Dublerq/rpi-lcd-menu* [18]. Tras la revisión de la solución y la realización de varias pruebas sobre nuestro dispositivo, llegamos a la conclusión de que merecia la pena la inversión de tiempo en el análisis del código y en el posterior desarrollo para su adaptación a las necesidades de nuestro proyecto.

Encontramos una librería Python desarrollada por Adafruit [19] que contaba con varias funciones ya implementadas para el control de un display como el 2004A utilizado y sustituimos la parte del desarrollo de **rpi-lcd-menu** que configuraba y manejaba el display por un objeto de la librería **CircuitPython_CharLCD**. Esta librería además formaba parte del proyecto de API CicuitPython, ya mencionado anteriormente y que habíamos utilizado en el desarrollo de las pruebas de funcionamiento del Keypad.

Llevamos a cabo varias modificaciones y la reescritura completa de algunos métodos, especialmente los que habían sido definidos para trabajar con dos líneas (display 16x2). Tras las modificaciones realizadas al proyecto rpi-lcd-menu conseguimos contar con un código importable en nuestro proyecto que nos permitiría crear un menú multinivel formado por el siguiente tipo de elementos y realizar un scroll circular a todos ellos si contuviesen mas de 4 subelementos:

- **Function_item**: elemento del menu que llama a una función (método).
- **Menu_item**: objeto padre de los elementos del menú.
- **Message_item**: elemento que imprime un mensaje en el display con la posibilidad de hacer scroll sobre este si ocupa más de 4 líneas.
- **Submenú_item**: objeto hijo de la clase principal que representa un nuevo menú dependiente del padre y al que se le podrán añadir los elementos antes vistos.

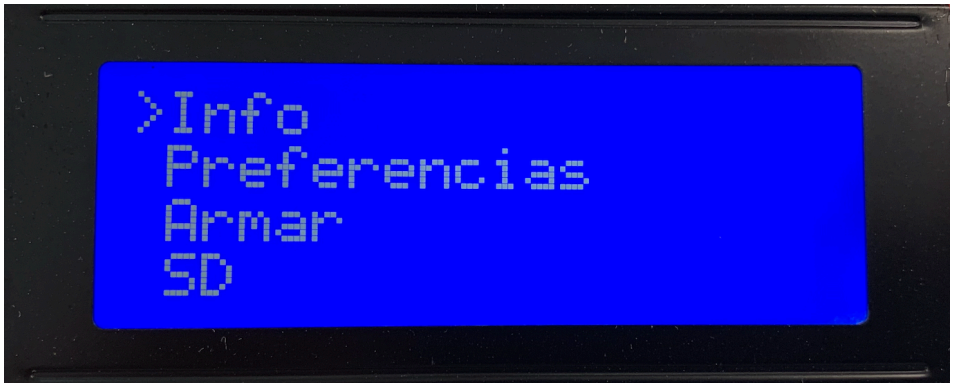

Figura 37. Menú Display Conpidential

Modificado el código y definidas las clases que conformarían el menú, integramos en el proyecto el código que habíamos desarrollado para el control del **Rotary Encoder**. Adaptando dicho código conseguimos hacer posible la navegación por las opciones del menú utilizando este elemento.

7.2 Autenticación 3FA

El siguiente punto en el que nos centramos fue el desarrollo del código que nos permitiese llevar a cabo el proceso de autenticación 3FA. Dado que habíamos desarrollado un pequeño ejemplo de autenticación basada en un PIN durante el análisis y pruebas del Keypad, reutilizamos dicho código para establecer el primer factor de autenticación, integrando el display LCD como medio de salida de información. De cara a combinar seguridad con practicidad a la hora de introducir un PIN, desarrollamos una funcionalidad que consistía en mostrar el digito introducido durante medio segundo y acto seguido sustituirlo por un *. De esta forma, el usuario podría comprobar que el digito tecleado era el correcto pero el PIN quedaría oculto en pantalla al instante minimizando riesgos.

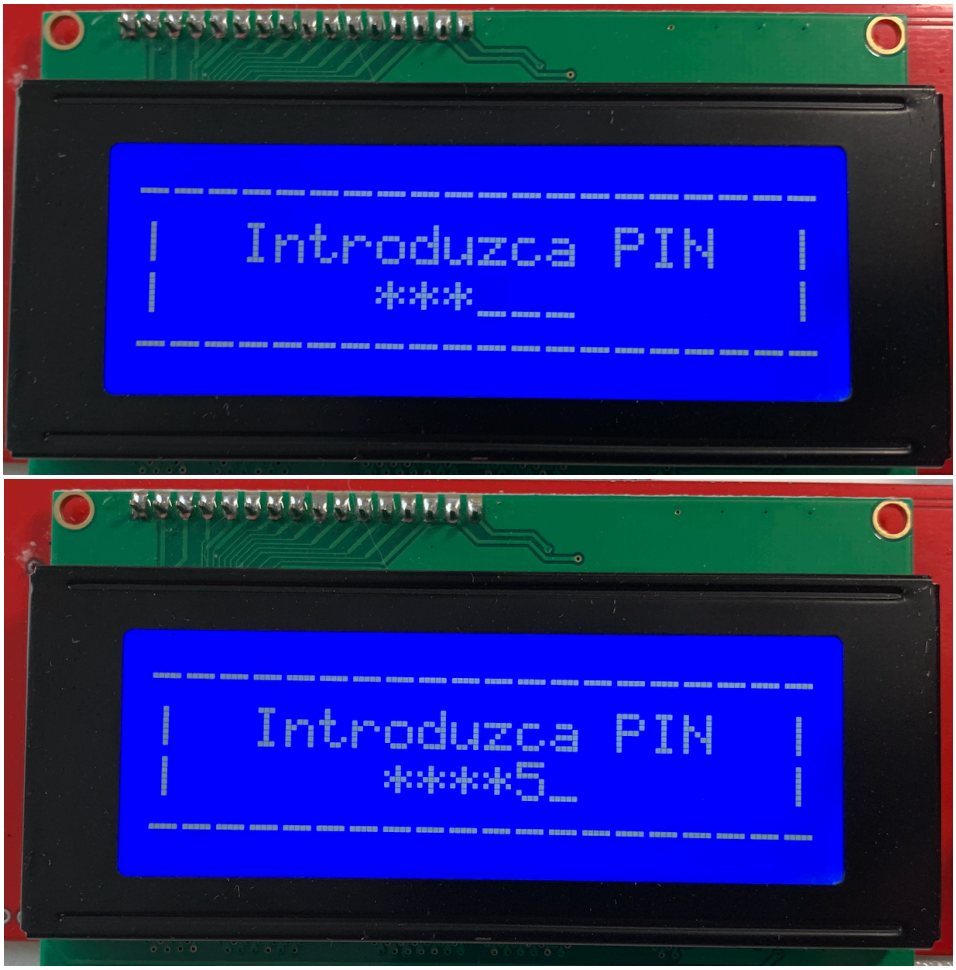

Figura 38. Ocultación del PIN

El primer factor de autenticación, "*algo que sé*", quedaría por tanto representado por un PIN de 6 dígitos que el usuario conocería. Como indicábamos en el capítulo 4, de cara a evitar el almacenamiento del PIN en plano o codificado, se decidió trabajar con la huella SHA256 correspondiente. En el prototipo del dispositivo ConPidential, este hash se encuentra almacenado en un fichero **auth.py** y es importado como constante por parte del módulo que lo necesita. Esta solución sería sustituida por otras opciones, como por ejemplo el almacenamiento del PIN en una BBDD, en un hipotético modelo final del dispositivo.

Tal y como se comentó en el capítulo 2, el segundo factor de autenticación, "*algo que tengo*", estaría basado en un TOTP (Time-based One-Time Password). Tras el análisis de las diferentes opciones existentes para la gestión de los TOTP y la integración de dicho mecanismo en nuestro proyecto, se optó por la utilización de la librería PyOTP [20]. Esta librería permitía integrar en nuestro desarrollo la generación y verificación de TOTPs y además era compatible con aplicaciones tales como Google Authenticator, 1Password, etc. dado que contaba con la funcionalidad de generar links **otpauth://** y convertirlos en códigos QR que podrían ser escaneados por dichas soluciones.

El primer paso necesario para la integración de PyOTP en nuestro desarrollo era la instalación de la librería correspondiente en el entorno Python:

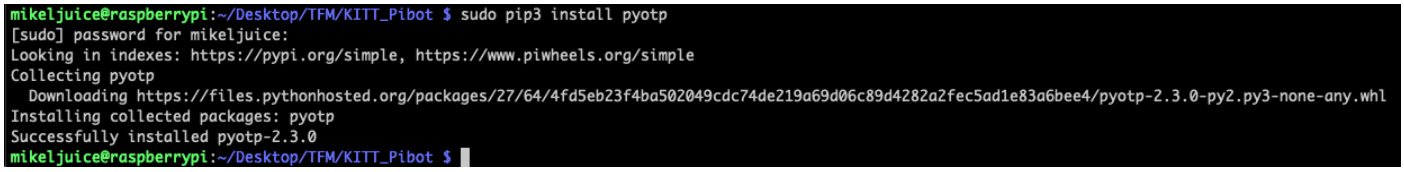

Figura 39. Instalación PyOTP

Tras la instalación de la librería y la revisión de la documentación correspondiente, seguimos el procedimiento indicado para la generación de un TOTP. En primer lugar, se debía generar una Secret Key codificada en base32 que serviría como base para la generación de nuestros tokens y para ello utilizaríamos la función **random_base32()** del módulo pyotp. Con el objetivo de poder gestionar la generación de TOTPs desde una aplicación como 1Password, deberíamos generar un link **otpauth://**. PyOTP nos facilita dicha gestión mediante el método **TOTP(base32_Key).provisioning_uri(cuenta_de_usuario, issuer_name=nombre_de_la_aplicación/servicio)**.

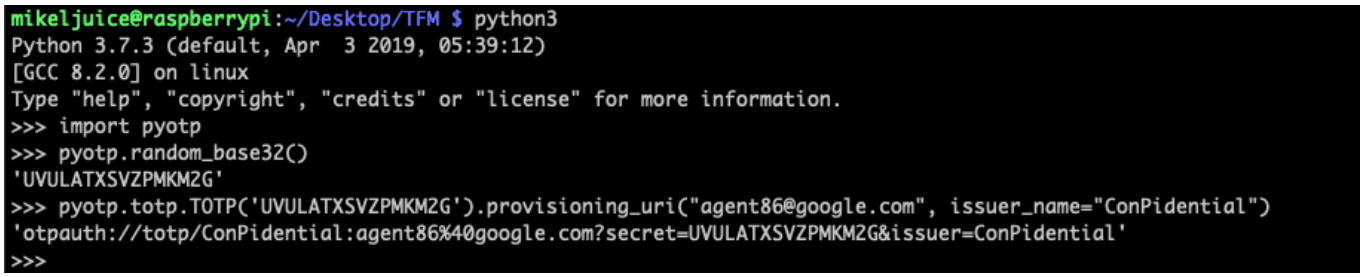

Figura 40. Generación de Secret Key y link otpauth

Una vez generada la URI otpauth, de cara a comprobar su compatibilidad teníamos dos opciones, introducir dicha URI de forma manual en una aplicación de gestión de TOTPs que permitiese ese tipo de introducción o generar el QR correspondiente para comprobar que cualquier aplicación de las antes comentadas era compatible. Optamos por esta última opción y para ello utilizamos la librería PyQRCode [21].

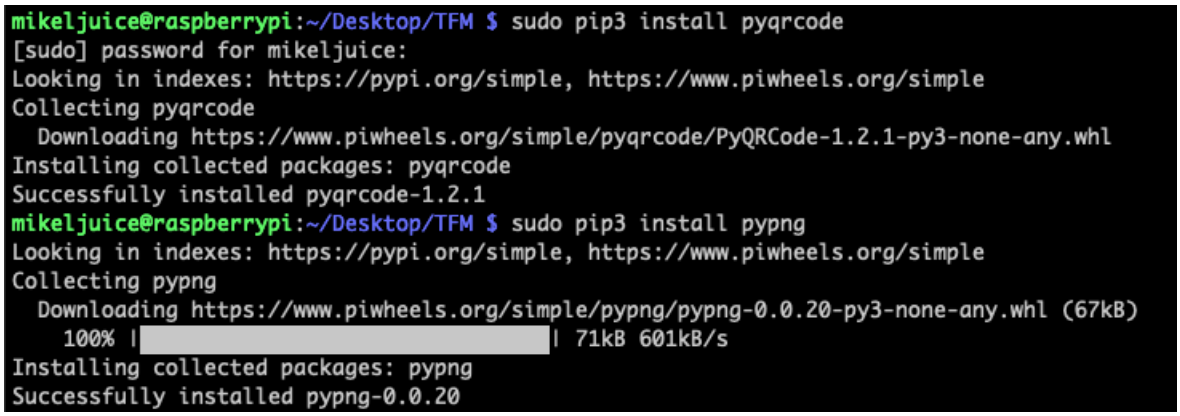

Figura 41. Instalación librería PyQRCode

Utilizando dicha librería desarrollamos un pequeño script llamado generateQR.py el cual crearía una imagen PNG con el código QR correspondiente al enlace enviado como parámetro. En este caso el parámetro sería la URI resultante del paso anterior. El resultado de la ejecución del script generateQR.py fue el siguiente:

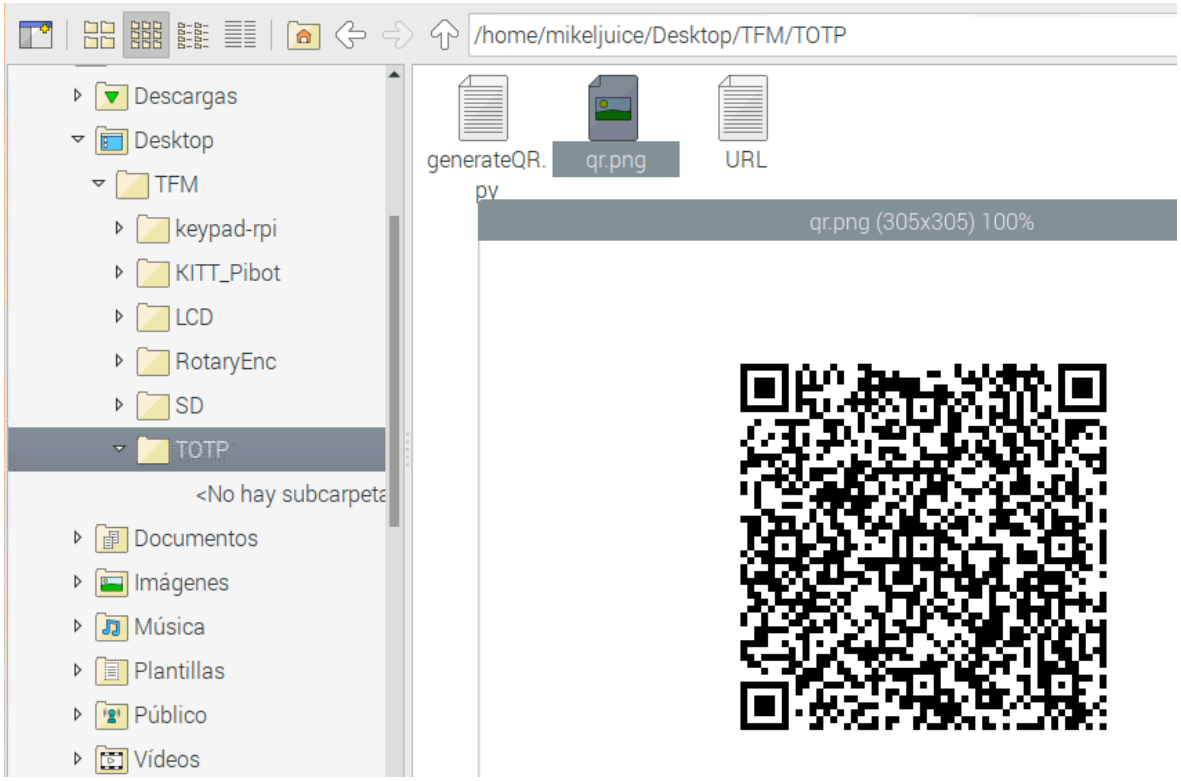

Figura 42. Código QR otpauth

El siguiente paso era cargar dicho QR en nuestra aplicación de gestión de TOTPs, en nuestro caso 1Password, y comprobar que los tokens generados por dicha aplicación eran validados por nuestro dispositivo ConPidential.

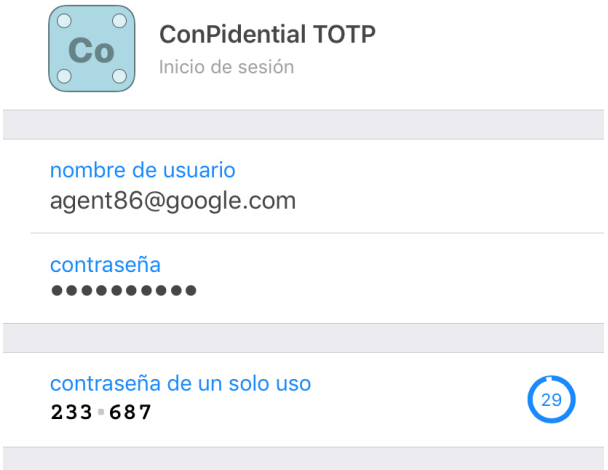

Figura 43. Generador TOTPs

Tras cargar correctamente el código QR en 1Password, utilizando nuestra Secret Key creamos un objeto pyotp y procedimos a lanzar la prueba de verificación del TOTP 233687 generado por 1Password. Como podemos observar a continuación, la verificación fue correcta y tras realizar una prueba 1 minuto después con el mismo código, comprobamos que este fue rechazado y por tanto la sincronización era correcta.

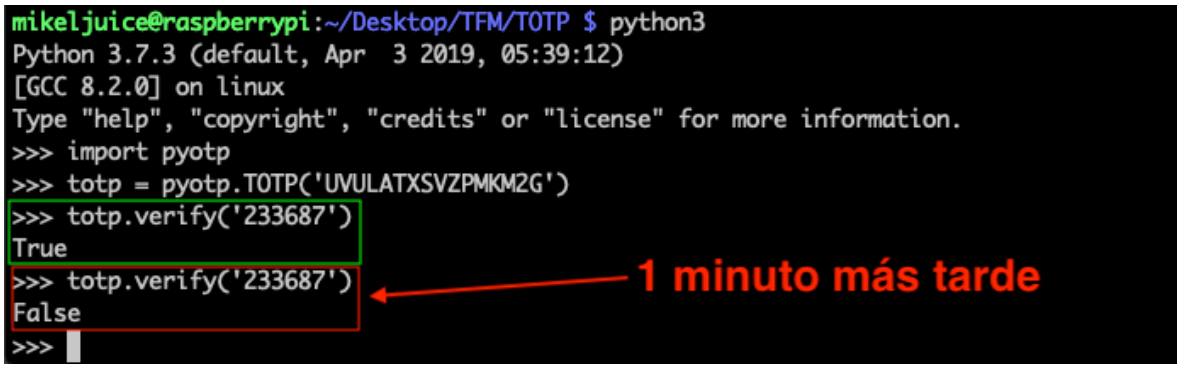

Figura 44. Comprobación código TOTP

Habiendo quedado validada esta funcionalidad, procedimos al desarrollo de un módulo llamado **totp.py** que facilitaba al resto del proyecto las funciones necesarias para comprobar la validez de los códigos TOTP entregados al dispositivo ConPidential. La Secret Key generada durante las pruebas fue almacenada en un fichero **codes.py** y definido como constante para que fuese importado por el código principal del módulo **totp.py**. Del mismo modo que indicábamos en el punto anterior, en una hipotética evolución del dispositivo, la Secret Key también debería ser almacenada, por ejemplo, en una BBDD.

Desarrolladas las funcionalidades necesarias para la gestión de los factores de autenticación 1 y 2, debíamos afrontar el desarrollo del 3er factor, "*algo que soy*". Como vimos en el capítulo 4, durante el análisis y pruebas del lector de huellas digitales llegamos a desarrollar código que nos permitía registrar una huella en la memoria del lector y llevar a cabo la verificación de la existencia de una determinada huella. La base del tercer factor estaba por tanto preparada y sólo quedaba integrarla en el código del proyecto ConPidential y añadir nuevas funcionalidades como el borrado completo de la memoria (eliminación de cualquier rastro de huella de la memoria del lector).

Una vez finalizada la integración de los 3 factores en nuestro código, llevamos a cabo un análisis sobre en qué funcionalidades del dispositivo íbamos a hacer uso de la autenticación 3FA. Identificamos dos tipos de acciones: acciones que no implicaban cambios relevantes en la solución ConPidential y acciones que conllevaban cambios importantes en su estado. Para toda acción que implicase un cambio de estado, que veremos más adelante, se requeriría la utilización del 3FA. Para el resto de acciones se utilizaría una autenticación básica por medio de un único factor: la huella digital.

Como hemos indicado, se incluyeron en el código las funciones necesarias para facilitar el borrado de la memoria del lector de huellas. Dado que esta funcionalidad iba a

ser habilitada para ser utilizada por el usuario, podría darse la circunstancia de que ConPidential tuviese que lanzar la autenticación básica o el 3FA y la memoria del lector se encontrase vacía. Para contemplar este caso, se desarrolló una función que comprobaba el estado de la memoria del lector y en el caso de que no existiesen huellas registradas, obviaría el tercer factor en el proceso 3FA y sustituiría la autenticación por huella en la autenticación básica por la introducción del PIN del usuario.

De cara a no tener que requerir el 3FA al usuario en cada acción relevante que llevase a cabo, se pensó en definir un TTL para la autenticación. Una vez autenticado mediante 3FA, el usuario dispondría de X minutos (5 en las PoC) para realizar acciones que requisieran de autenticación 3FA sin que se le volviese a requerir dicho nivel de autenticación.

Relacionado con el nivel de autenticación básico, se pensó en definir también un TTL pero en este caso se trataba del caso contrario. Se definiría un tiempo máximo de inactividad (interacción con el menú de la aplicación), superado el cual ConPidential bloquearía la interacción con el usuario y esta sería desbloqueada mediante la utilización del nivel de autenticación básico.

Aparecerían por tanto dos estados que ConPidential debería gestionar:

- Autenticado mediante 3FA:
  - "token" caducado
  - "token" no caducado
- Dispositivo bloqueado por inactividad

7.3 Estados del dispositivo

En el punto anterior hacíamos referencia a dos estados que el desarrollo debería controlar, pero no son los dos únicos estados definidos durante el diseño de la solución. A continuación, se procede a describir los diferentes estados de la solución ConPidential:

- **SD Enabled (True/False):** Este estado indica si el dispositivo tiene o no habilitado el lector de tarjetas SD de la Smart Controller. Como veremos a continuación, el usuario podrá modificar este estado y dado que se considera una modificación de bajo impacto, requerirá de autenticación básica.
- **Armed (True/False):** El estado "Armado" indica que el dispositivo cuenta con información cifrada y el control de la recepción de pruebas de vida ha sido activado. Se trata también de un estado que podrá ser modificado por el usuario y para el que será necesario estar autenticado mediante 3FA.

- **Encrypted_Files (lista):** En este caso no se trata de un estado definido por un booleano, sino por una lista. Dicha lista se encontrará vacia en caso de que no se haya almacenado y cifrado información alguna en el dispositivo ConPidential. En caso de que el dispositivo contenga información cargada por el usuario, esta lista contará con una entrada por cada fichero añadido con información relevante del mismo.

Estos tres registros de estado son críticos para el correcto funcionamiento de la solución y por tanto debían ser persistentes. Un cierre de la aplicación, un apagado o reinicio del dispositivo, etc. no debería afectar a esta información que debería ser recuperada en la reactivación del servicio. Tras un análisis de las opciones existentes para llevar a cabo este control de persistencia de la información, se probaron y descartaron algunas y finalmente se optó por la serialización de objetos basada en **pickle** [22]. Mediante este mecanismo podríamos llevar a cabo un dump binario a un fichero del estado de las variables/objetos que necesitásemos y nos asegurábamos de que podríamos restaurar el estado de dichas variables/objetos en el arranque de la aplicación sin problemas de conversión de tipos.

7.4 Menú de gestión del dispositivo ConPidential

Habiendo descrito los procedimientos y mecanismos para la autenticación y definidos los estados controlados por el dispositivo, contamos con la información necesaria para describir las funcionalidades accesibles desde el menú de gestión desarrollado para el prototipo.

A continuación, se presenta un esquema de la estructura del menú desarrollado:

```
- Info
- Preferencias - Atras
               - Lector de huellas - Atras
                                   - Registrar huella
                                   - Eliminar huellas
- Armar / Desarmar
- SD - Atras
     - Desactivar / Activar SD
     - Listar SD
     - Contenido - Atras
                 - Fichero_1 - File Info
                             - Almacenar
                 - Fichero_2 - File Info
                             - Almacenar
- Apagar
```

Figura 45. Esquema menú ConPidential

7.4.1 Info

Esta opción muestra en el display el siguiente texto scrollable:

```
--------------------
|   ConPidential   |
|     v. 0.007     |
|       2019       |
|    @mikeljuice   |
|     TFM MCYSI    |
|       UCLM       |
--------------------
```

7.4.2 Preferencias – Lector de huellas – Registrar huella

Tras comprobar si el usuario se encuentra autenticado mediante el 3FA, el sistema permite almacenar una nueva huella en la memoria del dispositivo lector de huellas.

7.4.3 Preferencias – Lector de huellas – Eliminar huellas

Tras comprobar si el usuario se encuentra autenticado mediante el 3FA, el sistema lanza un borrado completo de la BBDD de huellas almacenadas en la memoria del dispositivo lector de huellas.

7.4.4 Armar / Desarmar

En caso de que el sistema se encuentre “desarmado”, el menú mostrará la opción **Armar**. Esta opción comprobará si el usuario se encuentra autenticado mediante 3FA. En caso de que no lo esté, lanzará el proceso de autenticación 3FA. Una vez autenticado, comprobará si existen ficheros almacenados en el Vault. En caso de que existan, establecerá el estado “Armado” en el cual se comenzará a controlar desde ese instante la recepción de pruebas de vida por parte del usuario y no será posible añadir nuevos ficheros al Vault. En caso contrario, informará al usuario de que estando el Vault vacio no es posible armar el dispositivo.

En caso de que el sistema se encuentre “Armado”, el menú mostrará la opción **Desarmar.** Esta opción comprobará si el usuario se encuentra autenticado mediante 3FA. En caso de que no lo esté, lanzará el proceso de autenticación 3FA. Una vez autenticado, establecerá el estado “Desarmado” en el cual es posible añadir nuevos ficheros al Vault y el dispositivo dejará de supervisar la recepción de pruebas de vida por parte del usuario.

7.4.5 SD - Desactivar / Activar SD

En caso de que ConPidential tenga marcado como **Disabled** el lector de tarjetas SD del Smart Controller, el menú mostrará la opción **Activar SD**. Esta opción lanzará una función que llamará al sistema y ejecutará con privilegios de administrador el comando necesario para cargar en memoria el driver **mmc_spi**, habilitando de esta forma el lector. Almacenará en el fichero de persistencia (pickle) el estado del lector SD como **Enabled** y modificará el texto de la opción del menú a "Desactivar SD".

En caso de que ConPidential tenga marcado como **Enabled** el lector de tarjetas SD, el menú mostrará la opción **Desactivar SD**. Esta opción lanzará una función que llamará al sistema y ejecutará con privilegios de administrador los comandos necesarios para desmontar la tarjeta SD, en caso de estar montada en **/media/SD_EXT**, y para descargar de memoria el driver **mmc_spi**, deshabilitando así el lector. Almacenará en el fichero de persistencia (pickle) el estado del lector SD como **Disabled** y modificará el texto de la opción del menú a "Activar SD".

7.4.6 SD – Listar SD

Esta opción que aparecerá únicamente en caso de que el lector de tarjetas se encuentre habilitado, comprobará si existe una tarjeta SD montada en el sistema y analizará el directorio raíz montado en **/media/SD_EXT** en busca de ficheros. En caso de que los haya, por cada uno de ellos registrará su nombre, su tamaño en Bytes y creará una nueva entrada **Nombre_Fichero** en el submenú **SD – Contenido** ligado a este fichero.

7.4.7 SD – Contenido – Nombre_Fichero

El submenú Contenido aparecerá únicamente en caso de que le lector se encuentre habilitado y una vez se haya lanzado la opción "Listar SD". Como se ha indicado, por cada fichero se creará una entrada en el submenú **Contenido** que realmente será un nuevo submenú identificado por el nombre del fichero en cuestión (primeros 19 caracteres del nombre). Dicho submenú contará con dos entradas **File Info** y **Almacenar**.

7.4.7.1 SD – Contenido – Nombre_Fichero – File Info

Esta opción mostrará un texto scrollable en el display en el que aparecerá el nombre completo del fichero y su tamaño en Bytes.

7.4.7.2 SD – Contenido – Nombre_Fichero – Almacenar

Esta opción comprobará en primer lugar si el dispositivo Conpidential se encuentra **Armado.** En caso de estarlo, informará al usuario de que no es posible añadir nuevos

archivos al vault estando el dispositivo en modo Armado. En caso de que el dispositivo se encuentre **Desarmado**, se comprobará si el usuario se encuentra autenticado mediante 3FA, lanzando el proceso de autenticación en caso de que no lo esté.

Tras la verificación de la autenticación, Conpidential comprobará si el fichero seleccionado ya se encuentra almacenado en el Vault. Para ello, calculará el SHA256 del fichero seleccionado y lo cotejará con el campo correspondiente a SHA256_origen de la lista de ficheros cifrados y almacenados en el Vault. En caso de que el SHA256 coincida con alguna de las entradas de la lista de ficheros almacenados, informará al usuario de que el fichero ya se encuentra en el Vault. En caso contrario, procederá al cifrado y almacenado del fichero.

El proceso de cifrado y almacenado de un fichero consta de los siguientes pasos:

1. Cálculo del SHA256 del fichero.
2. Generación de un string aleatorio (utilizando funcionalidad Zymkey) con el que nombrar el fichero destino, minimizando la información que podría ser recopilada de dicho fichero.
3. Cifrado y firmado del fichero origen mediante las claves almacenadas en el módulo Zymkey.
4. Almacenado del fichero resultante en el directorio correspondiente al Crypted Vault utilizando como nombre el string obtenido en el paso 2.
5. Almacenado de los datos “Nombre fichero original”, “SHA256”, “Nombre fichero cifrado” en la lista de ficheros cifrados y almacenado de la lista completa en el fichero de persistencia correspondiente (pickle).

7.4.8 Apagar

Esta opción finaliza todos los procesos lanzados por la aplicación y libera los recursos GPIO utilizados. Antes de finalizar con los procesos requiere confirmación mediante la lectura de la huella digital. En caso de que no existan huellas registradas en el lector, se requerirá la introducción del PIN del usuario.

7.5 Control del envío y recepción de pruebas de vida

Retomamos en este punto el bot de Telegram desarrollado durante la fase vista en el capítulo 6. El desarrollo inicial del bot contemplaba la restricción de respuesta al lanzamiento de comandos a una lista de usuarios definidos en la configuración de la solución (en el prototipo, dicha lista se encuentra hardcodeada) y contaba con dos comandos desarrollados: **/start** y **/test**

Tras definir las necesidades y los casos de uso del proyecto, avanzamos en el desarrollo del bot añadiéndole al mismo varias funcionalidades. A continuación, definimos los estados propios de este módulo y las funcionalidades desarrolladas para la gestión de las pruebas de vida.

@KITT_Pibot cuenta con dos estados principales:

- **Armed_bot (True/False)**: Este estado está enlazado con el estado Armed visto en el apartado 7.3. Cuando el dispositivo es armado o desarmado, se notifica al bot el cambio de estado y éste lo refleja en una variable de estado propia.
- **Released (True/False)**: Este estado es True si el tiempo de espera entre recepción de pruebas de vida ha sido sobrepasado y la información ha sido revelada.

El bot es el encargado de, mediante un hilo independiente, controlar el TTL definido en la configuración para la vigencia de una prueba de vida recibida. Se desarrolló un método basado en un bucle infinito que comprueba continuamente el estado del deadline (fin del TTL) y si se han recibido pruebas de vida. En caso de que se agote el TTL sin haber recibido una nueva prueba de vida, este hilo será el encargado de llamar a la función **release_the_kraken** que, como veremos a continuación, llevará acabo las tareas necesarias para revelar la información almacenada.

¿Cómo se lleva a cabo la validación de pruebas de vida recibidas? Para este proceso se desarrolló un método encargado de gestionar el comando **/imalive** añadido a nuestro bot. De cara a controlar la veracidad y la autoría del mensaje (dentro de unos límites), se decidió que el comando **/imalive** debía ir acompañado de un token que no sería otro que el TOTP definido para el mecanismo de autenticación 3FA. De esta forma, evitaríamos que, ante la interceptación de un mensaje, se pudiese obtener información relevante que pudiese ser utilizada para reproducir el efecto tras el reenvío del mensaje por un tercero que nos hubiera suplantado.

Cuando el bot recibe una prueba de vida mediante el comando "**/imalive TOTP**" se lanza la validación del TOTP y en caso de ser válido se resetea el TTL correspondiente y se informa al hilo encargado de chequear el deadline que ha sido recibida una nueva prueba de vida válida. En caso de recibirse un TOTP no válido, se informará al usuario y se le permitirá volver a enviar un nuevo código hasta un máximo de 4 errores. En caso de recibir el 4º error, el bot bloqueará la validación de nuevas pruebas de vida durante un periodo de tiempo establecido (5 minutos en el prototipo).

Figura 46. Comando /imalive TOTP

En el caso de que el dispositivo y por tanto el bot se encuentren desarmados, el hilo que verifica los deadlines no realizará acción alguna y los comandos **/imalive** enviados al bot serán respondidos informando de que el dispositivo se encuentra desarmado.

Figura 47. Info ConPidential desarmado

De cara a facilitar al usuario el control del TTL de la última prueba de vida enviada, se desarrolló un método encargado de gestionar el comando **/info.** Este comando responde al usuario indicando el estado del dispositivo (Armado – Desarmado) y le indica cuando fue recibida la última prueba de vida válida y cuando finaliza el TTL de la misma.

Figura 48. Comando /info

Además de las respuestas a los comandos enviados por un determinado usuario, se pensó incluir una funcionalidad que permitiese realizar envíos de notificaciones a todos los usuarios que formasen parte de la lista de usuarios autorizados. De este modo, ante un cambio relevante en el estado del dispositivo como puede ser el arme – desarme, el/los usuarios serían avisados. En combinación con la funcionalidad de detección de eventos de posible manipulación facilitada por el módulo Zymkey, desarrollamos un método que ante la recepción de una notificación de evento TAP por parte de Zymkey enviase mediante Telegram una notificación a los usuarios autorizados alertándoles de dicho evento.

Figura 49. Notificación dispositivo armado

Figura 50.Notificación evento TAP

Para finalizar con la descripción de las gestiones realizadas por el bot, llegamos a la función **release_the_kraken**. Como hemos indicado anteriormente, esta función será llamada por el hilo que controla los TTLs de las pruebas de vida recibidas, en caso de que el TTL haya sido agotado y no se haya recibido una nueva prueba de vida. La finalidad de dicha función es revelar la información almacenada en el dispositivo. Dado que se llevan a cabo varias acciones hasta llegar a dicho objetivo, las describiremos a continuación con mayor detalle.

El proceso comienza con el descifrado de los ficheros almacenados en el Crypted Vault. Como hemos indicado anteriormente, los ficheros son cifrados y firmados por el módulo Zymkey y el nombre del fichero es sustituido por un string aleatorio. Para llevar a cabo el proceso de descifrado, por cada fichero existente en el Crypted Vault, el primer paso es utilizar el módulo Zymkey para llevar a cabo el descifrado y verificación de firma del fichero. Como segundo método de verificación de integridad de la información, obtenemos el SHA256 del fichero descifrado por Zymkey y lo comparamos con el SHA256 del fichero original que habíamos almacenado en memoria. Si ambos hashes coinciden, damos por válido el fichero, lo renombramos con su nombre original que mantenemos también en memoria, lo almacenamos en la Release Vault y escribimos una nueva línea con el nombre del fichero final y su SHA256 en un fichero digest que generamos en el Release Vault. En caso de que los hashes no coincidan, el descifrado haya fallado o el fichero no se encontrase en la lista Encrypted_files cargada en memoria, dicho fichero no será enviado al Release Vault.

Tras finalizar con el proceso de descifrado y validación de cada uno de los ficheros existentes en el Crypted Vault, procedemos a firmar mediante una de las claves privadas de Zymkey el fichero de digest que contiene la relación de ficheros revelados y sus hashes, de forma que se pueda comprobar la integridad de los ficheros revelados cuando sean recibidos por los destinatarios configurados. Para que los destinatarios puedan comprobar la validez de la firma del fichero digest, se incluye el fichero PEM de la clave pública correspondiente a la clave privada con la que ha sido firmado dicho fichero.

Por último, tras verificar que el proceso de envío de la información ha finalizado correctamente, se llevará a cabo un borrado de toda la información existente tanto en el Crypted Vault como en el Release Vault y se modificará el estado del dispositivo y por tanto del bot a Armed = False y Released = True.

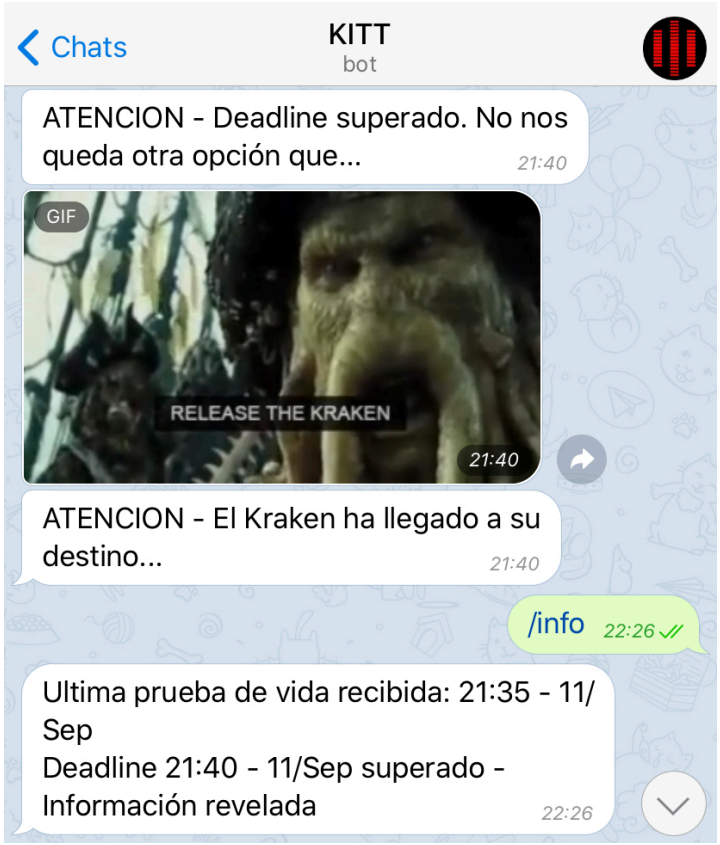

Figura 51. Release the Kraken

7.6 Revelación de la información

En el punto anterior, hacíamos mención al envío de la información revelada a los destinatarios establecidos en la configuración. Como indicábamos en el capitulo 2, se realizó un primer análisis de las posibilidades de envío/publicación de la información a revelar. Las opciones eran muy diversas y comprendían desde la subida de la información a un FTP al desarrollo de una solución web de gestión de archivos, pasando por el envío de la información revelada por medio de un correo electrónico.

Tras analizar las ventajas e inconvenientes de cada una de las opciones y de valorar el esfuezo que implicarían, decidimos optar por el envío de la información por medio de un correo eletrónico. Con la idea de desarrollar un módulo encargado del envío de correos electrónicos que cumpliese con unos mínimos respecto a la seguridad, analizamos las posibilidades que nos permitía la API de Gmail.

Comprobamos que mediante la API podíamos utilizar OAuth para pedir a un usuario de Google privilegios de envío de correo desde su cuenta de Gmail. Tras la aceptación de dicho privilegio por parte del usuario obtendríamos un token OAuth que nos permitiría realizar envíos de correos electrónicos directamente mediante la API, sin necesidad de

establecer conexiones inseguras contra un SMTP o de almacenar credenciales de cuentas de correo electrónico.

La forma más sencilla de llevar a cabo la configuración del proyecto en Google y automatizar en cierta medida el proceso de obtención de las autorizaciones necesarias por parte del usuario, era utilizar Python Quickstart [23]:

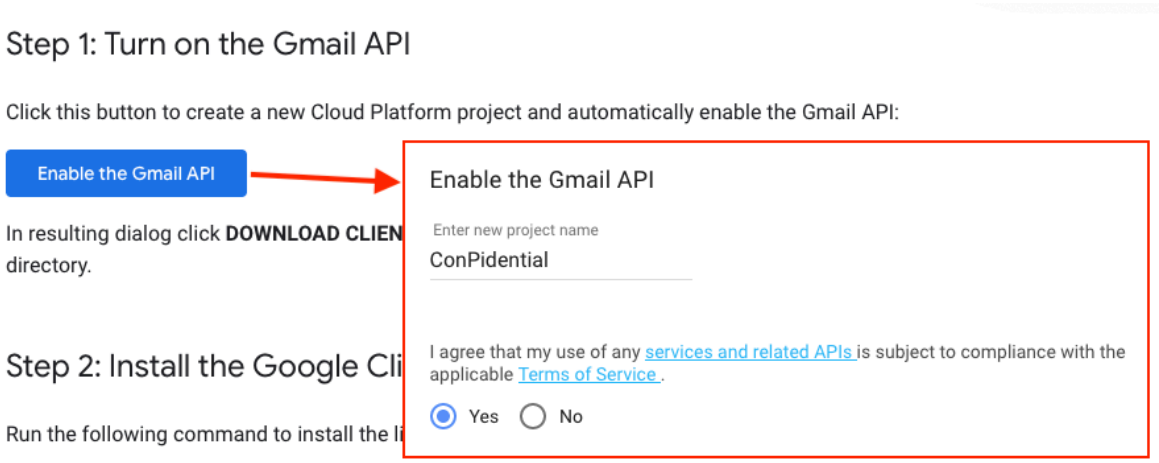

Figura 52. Gmail Python Quickstart_1

Tras crear un nuevo proyecto, el sistema nos devolverá un ID de cliente y un password para establecer la comunicación e identificarnos con la API de Gmail. En nuestro caso, descargamos el fichero de configuración de cliente ya que dicho fichero sería requerido más adelante.

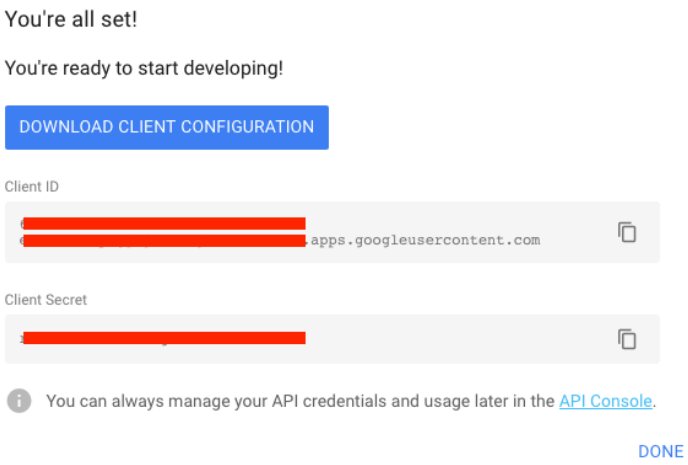

Figura 53. Gmail Python Quickstart_2

El siguiente paso consistía en la instalación de las librerías Python necesarias para manejar la API desde nuestro código.

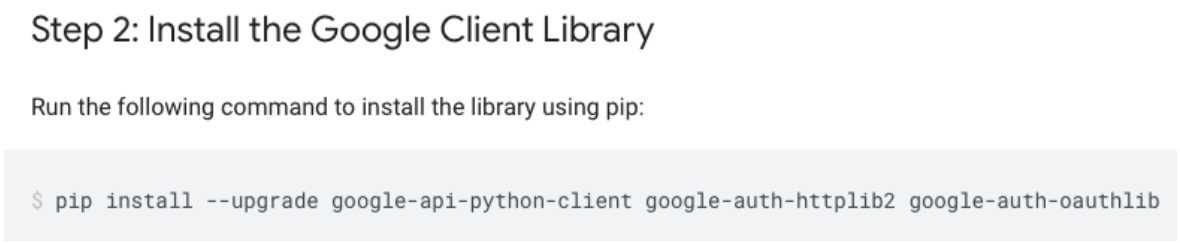

Figura 54. Gmail Python Quickstart_3

Una vez instaladas las librerías, podríamos descargar y hacer uso del script Python **quickstart.py** para el que necesitaríamos tener en el mismo path el fichero **credentials.json** descargado en el paso anterior. Tras el lanzamiento del script, estando autenticados con el usuario Google al que deseábamos pedir los privilegios, se mostraron los mensajes de concesión de permisos correspondientes.

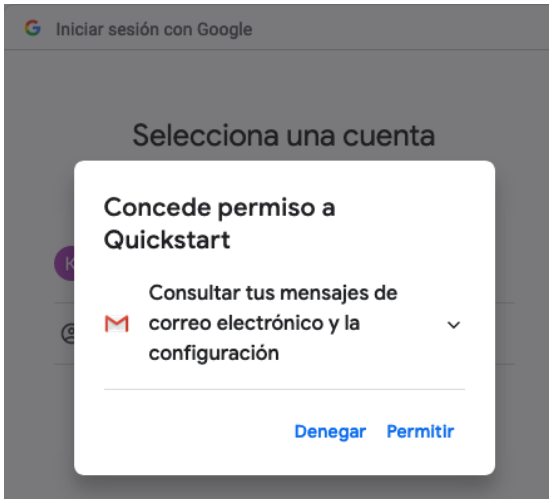

Figura 55. Gmail Python Quickstart_4

Tras la finalización del proceso de aceptación de permisos, el script **quickstart.py** creó el fichero **token.pickle** en el que se almacenaron los datos correspondientes al token OAuth generado.

Utilizando como base el fichero **quickstart.py** para obtener el código necesario para la gestión de los tokens OAuth utilizando como autenticación el fichero **credentials.json**, tras analizar los ejemplos de código para el envío de emails con adjuntos a través de la API que encontramos en la documentación de la API de Gmail [24], desarrollamos nuestro propio módulo Python (**gmail_ops.py**) que, tras comprobar la validez del token existente en el fichero **token.pickle**, prepararía un correo electrónico en formato MIME con los adjuntos que le indicásemos como parámetro y lo enviaría a los destinatarios indicados.

A continuación, mostramos una muestra del email enviado por el módulo ConPidential tras la no recepción de la prueba de vida correspondiente antes de superar el deadline:

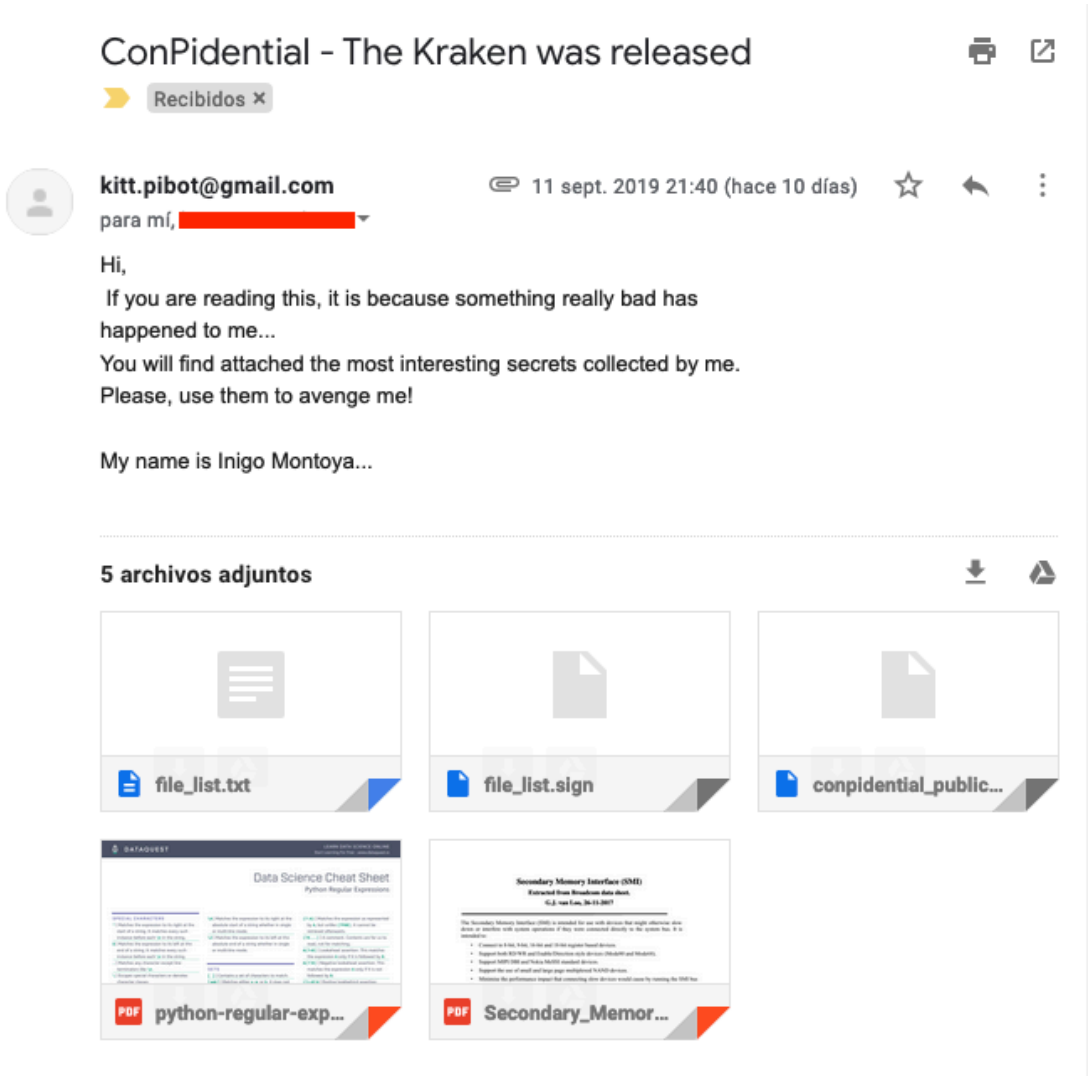

Figura 56. Correo Revelado información

CAPÍTULO 8. CONCLUSIONES Y PROPUESTAS

En este capítulo final, se presentan las conclusiones respecto al trabajo realizado, y se realizan algunas sugerencias para, en el futuro, sofisticar y completar una solución como la prototipada en este TFM.

8.1 Conclusiones

Con las tecnologías y dispotivos escogidos, fue posible llevar a cabo el proyecto y conseguir los objetivos establecidos. Cada uno de los objetivos específicos presentados en el capítulo 1, y que han guiado el desarrollo del trabajo, se cumplen en el resultado final "ConPidential", y con ello, creemos que el objetivo general se alcanza satisfactoriamente. Desde el punto de vista del autor de este TFM, con el trabajo realizado se ha logrado responder al reto personal establecido.

Aunque se trate de un primer prototipo, ConPidential pone a disposición del usuario una solución factible y razonablemente segura para custodiar información sensible mientras se reciba la prueba de vida en el periodo de tiempo especificado, y para divulgar la información a los contactos preestablecidos, en caso contrario.

Sin embargo, cabe señalar que a lo largo del desarrollo del trabajo se han detectado ciertas limitaciones, de las que somos conscientes, y se ha pensado en diversas posibilidades de mejora que no era posible abordar en un primer prototipo, por la limitación temporal que establecía la fecha de entrega de este trabajo. Tanto las limitaciones, como estas ideas para la sofisticación y mejora del dispositivo, nos permiten establecer líneas de desarrollo futuro que presentamos en siguiente epígrafe.

8.2 Trabajo futuro y posibles ampliaciones

Para finalizar con este último capítulo, detallamos algunas propuestas de trabajo futuro que surgen, como ya se ha indicado, del hecho de ser conscientes de ciertas limitaciones en el prototipo, o de ideas adicionales para diseñar y construir una solución más sofisticada y completa.

En primer lugar, una limitación del trabajo realizado es que el prototipo cuenta con una única vía de comunicación con internet, y por tanto, con el usuario. Dado que la comunicación con el usuario es crítica, y la revelación de la información se realiza por medio de internet, el dispositivo debería contar con un segundo medio de comunicación redundante, utilizable en caso de que fallara la primera vía. Para ello, podría incorporarse a una futura nueva versión del dispositivo ConPidential una conexión móvil 4G, además de la conexión WiFi o cableada con la que cuenta el dispositivo actualmente.

Además, habría que tener en cuenta que basamos la comunicación de la prueba de vida en un servicio de terceros, que como cualquier otro servicio online puede sufrir caídas de servicio. De nuevo es necesario destacar que esta comunicación es crítica, por lo que sería preciso desarrollar una vía de comunicación secundaria, lo más robusta posible, que podría ser por ejemplo la utilización de mensajes SMS.

Otro aspecto clave para incorporar a futuras versiones del dispositivo sería redoblar los esfuerzos para minimizar la posibilidad de una manipulación física del dispositivo. Una posible vía de avance, en este sentido, sería incorporar una solución ya disponible, propiedad del desarrollador del módulo ZYMKEY. Tal y como indicábamos en el capítulo 5 (pag. 44), esta solución se integra con el módulo y permite detectar cualquier tipo de manipulación física del dispositivo y reaccionar ante esta clase de evento borrando las claves de encriptación, de modo que el módulo queda completamente inservible.

Así mismo, es preciso ser consciente de que, en un escenario en el que la información es altamente sensible, el disponer de un único dispositivo en el que tenerla almacenada, hace que dicho dispositivo sea el objetivo principal de un potencial ataque que pretenda evitar que la información se divulgue: si alguien se hace con el dispositivo y elimina las opciones de conectividad, habría logrado su objetivo. Como solución a esta limitación, planteamos el desarrollo de una solución basada en varios nodos que se supervisen entre ellos, y compartan la información, de cara a descentralizar el punto de ataque y dar continuidad al servicio en caso de caída de alguno de los nodos.

Finalmente, es importante indicar que, al tratarse de un dispositivo diseñado como prototipo y no como producto final, no se han llevado a cabo tareas de hardening del SO que soporta la solución. Del mismo modo que hacemos referencia a elementos que nos permitirían aumentar la seguridad del entorno físico del dispositivo, sería crucial llevar a

cabo el hardening del SO y de las aplicaciones y/o servicios que corriesen sobre la release final del proyecto.

Con todo ello, creemos que hemos dado un primer paso para el desarrollo de un dispositivo que permitiera custodiar o, llegado el momento, revelar información altamente sensible, que sería útil tanto para los espías de la ficción que tanto nos entretienen, como para los -lamentablemente- necesarios "*whistleblowers*", personas que asumen grandes riesgos para dar a conocer a la ciudadanía fraudes o abusos que se cometen en distintas instancias.

BIBLIOGRAFÍA

[1] «HD44780 LCD User-Defined Graphics,» [En línea]. Available: https://www.quinapalus.com/hd44780udg.html.

[2] «HOW TO SETUP AN LCD ON THE RASPBERRY PI AND PROGRAM IT WITH PYTHON,» [En línea]. Available: http://www.circuitbasics.com/raspberry-pi-lcd-set-up-and-programming-in-python/.

[3] «Raspberry Pi – LCD display 1602 / 2004 via GPIO,» [En línea]. Available: http://domoticx.com/raspberry-pi-lcd-display-1602-2004-via-gpio/.

[4] «How to Use MMC/SDC,» 2018. [En línea]. Available: http://elm-chan.org/docs/mmc/mmc_e.html.

[5] B. V. Brown, «Adding a secondary SD card on Raspberry PI,» [En línea]. Available: https://ralimtek.com/raspberry%20pi/electronics/software/raspberry_pi_secondary_sd_card/.

[6] «Device Tree for DUMMIES,» 2013. [En línea]. Available: https://bootlin.com/pub/conferences/2013/elce/petazzoni-device-tree-dummies/petazzoni-device-tree-dummies.pdf.

[7] «Vonger's Blog,» 2014. [En línea]. Available: http://vonger.cn/?p=1045.

[8] J. Madieu, Linux Device Drivers Development: Develop customized drivers for embedded Linux, Packt Publishing, 2017.

[9] «Automount USB drives with systemd,» 2016. [En línea]. Available: https://serverfault.com/questions/766506/automount-usb-drives-with-systemd/767079#767079.

[10] «CircuitPython,» [En línea]. Available: https://learn.adafruit.com/circuitpython-on-raspberrypi-linux/overview.

[11] «How to use a Raspberry Pi Fingerprint Sensor for Authentication,» 2017. [En línea]. Available: https://tutorials-raspberrypi.com/how-to-use-raspberry-pi-fingerprint-sensor-authentication/.

[12] «HSM,» [En línea]. Available: https://es.wikipedia.org/wiki/HSM.

[13] «python-telegram-bot: Introduction to the API,» [En línea]. Available: https://github.com/python-telegram-bot/python-telegram-bot/wiki/Introduction-to-the-API.

[14] «Python-telegram-bot: Post a text message,» [En línea]. Available: https://github.com/python-telegram-bot/python-telegram-bot/wiki/Code-snippets#post-a-text-message.

[15] «Post a gif from a URL (send_animation),» [En línea]. Available: https://github.com/python-telegram-bot/python-telegram-bot/wiki/Code-snippets#post-a-gif-from-a-url-send_animation.

[16] «Restrict access to a handler,» [En línea]. Available: https://github.com/python-telegram-bot/python-telegram-bot/wiki/Code-snippets#restrict-access-to-a-handler-decorator.

[17] «Github: python-telegram-bot,» [En línea]. Available: https://github.com/python-telegram-bot/python-telegram-bot/wiki/Code-snippets#send-action-while-handling-command-decorator.

[18] «Github: rpi-lcd-menu,» [En línea]. Available: https://github.com/Dublerq/rpi-lcd-menu.

[19] «CircuitPython CharLCD Library,» [En línea]. Available: https://circuitpython.readthedocs.io/projects/charlcd/en/latest/api.html.

[20] «PyOTP - The Python One-Time Password Library,» [En línea]. Available: https://pyotp.readthedocs.io/en/latest/.

[21] «PyQRCode's documentation,» [En línea]. Available: https://pythonhosted.org/PyQRCode/.

[22] «Python documentation: Pickle Serialization,» [En línea]. Available: https://docs.python.org/3/library/pickle.html.

[23] «Python Quickstart: Gmail API,» [En línea]. Available: https://developers.google.com/gmail/api/quickstart/python.

[24] «Email Sending - Gmail API documentation,» [En línea]. Available: https://developers.google.com/gmail/api/guides/sending.

ARCHIVOS QUE SE ENTREGAN CON LA MEMORIA

Junto a la memoria, se envían para su evaluación los siguientes recursos:

- Código fuente del proyecto dentro del directorio ConPidential_src.
  - Ha sido eliminado todo rastro de tokens y credenciales utilizadas para el desarrollo de la PoC
  - La estructura de directorios que sigue el proyecto desarrollado es la siguiente:

```
├── adafruit_character_lcd.py
├── adafruit_matrixkeypad.py
├── config
│   ├── auth.py
│   └── screens.py
├── conpidential_menu.py
├── rpilcdmenu
│   ├── base_menu.py
│   ├── helpers
│   │   ├── __init__.py
│   │   └── text_helper.py
│   ├── __init__.py
│   ├── items
│   │   ├── function_item.py
│   │   ├── __init__.py
│   │   ├── menu_item.py
│   │   ├── message_item.py
│   │   ├── range_selection_item.py
│   │   ├── select_item.py
│   │   └── submenu_item.py
│   ├── rpi_lcd_hwd.py
│   ├── rpi_lcd_menu.py
│   ├── rpi_lcd_submenu.py
│   ├── selection_menu.py
│   ├── version.py
│   └── views
│       ├── __init__.py
│       └── message_view.py
├── telegram_bot_v3.py
├── three_factor_auth_mod.py
└── utils
    ├── config
    │   └── codes.py
    ├── fingerprint_ops.py
    ├── gmail_ops.py
    ├── __init__.py
    ├── sd_ops.py
    ├── totp.py
    └── zymkey_ops.py
```

- Documentos utilizados durante la fase de diseño y desarrollo, entre los que se encuentran la documentación de la API Python del módulo Zymkey o el diagrama electrónico del Smart Controller. Los cuales podemos encontrar en el directorio Anexos.
- Enlaces a los ficheros que forman parte de la defensa del TFM. Los cuales podemos encontrar en el directorio Defensa