1. Resolución

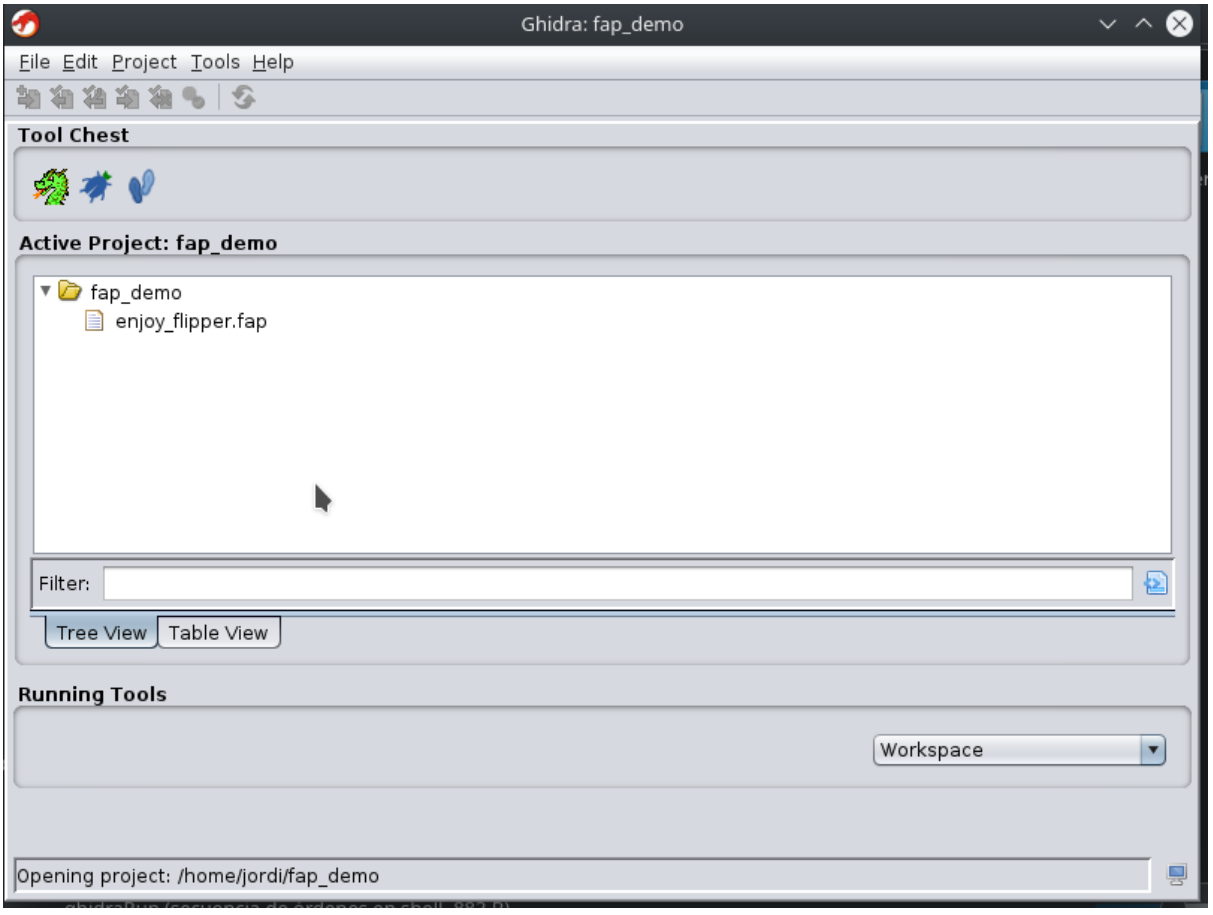
En el zip tenemos un fichero con extensión **.fap**. Si usamos el comando *file* nos informa de que se trata de un binario ejecutable para arquitectura ARM de 32 bits.

```
$> file enjoy_flipper.fap
enjoy_flipper.fap: ELF 32-bit LSB relocatable, ARM, EABI5 version 1 (SYSV), not stripped
```

Si hacemos un strings con el formato de la flag podemos ver que solo se encuentra un literal que no cumple el criterio de md5:

```
$> strings enjoy_flipper.fap | grep flagHunters{
flagHunters{esta_no_es_la_flag}
```

Pasamos al análisis estático con Ghidra para ver si podemos sacar más información.



Pasamos el CodeBrowser ( ) para ver el código decompilado y podemos observar que existen dos funciones que destacan: *decode_flag* y *enjoy_flipper_app*.

Por el nombre, *decode_flag*, parece clave para poder obtener la solucción. Si leemos el decompilado vemos que es un método sencillo:

Decompile: decode_flag - (enjoy_flipper.fap)

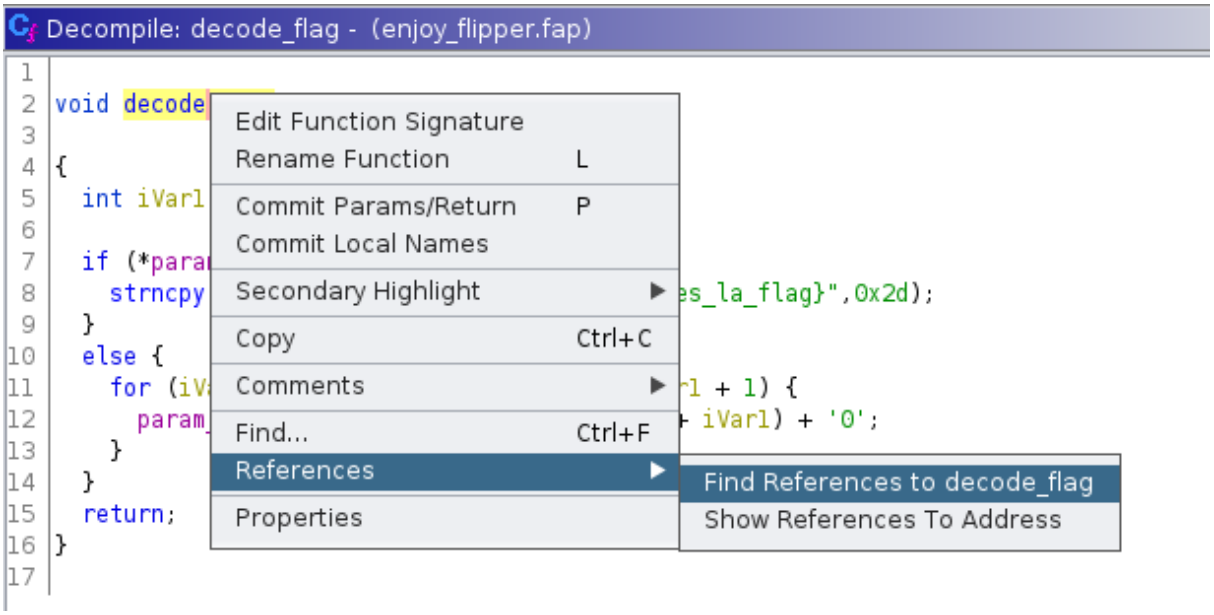
```
1
2  void decode_flag(char *param_1,int param_2)
3
4  {
5    int iVar1;
6
7    if (*param_1 == '\0') {
8      strncpy(param_1 + 1,"flagHunters{esta_no_es_la_flag}",0x2d);
9    }
10   else {
11     for (iVar1 = 0; iVar1 < 0x2d; iVar1 = iVar1 + 1) {
12       param_1[iVar1 + 1] = *(char *)(param_2 + iVar1) + '0';
13     }
14   }
15   return;
16 }
17
```

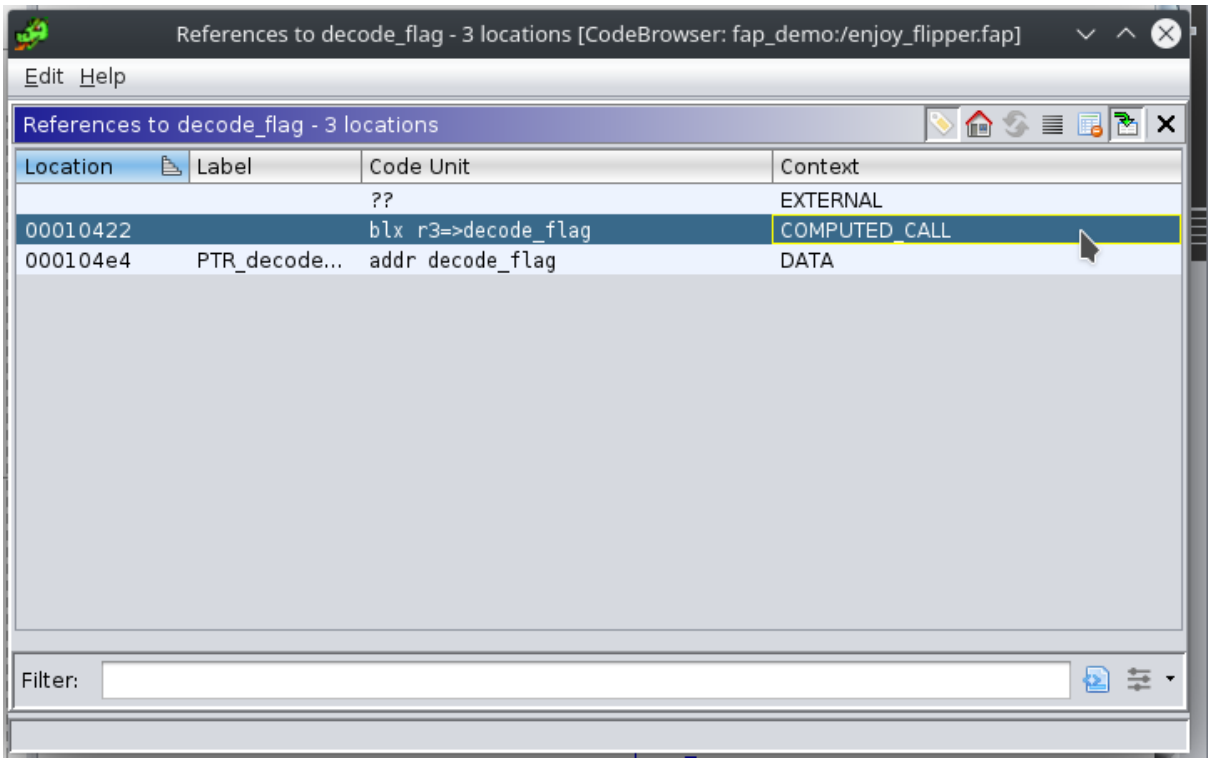
En resumen, recibe dos parámetros: el primero se usa para determinar si se copia o no la flag falsa o se procesa la auténtica, que por lo visto viene pasada por el parámetro 2. Si nos fijamos en el if y luego en la asignación, vemos que empieza a asignar por 1 y no por 0, seguramente porque se trate de una estructura y la primera posición es para determinar que ejecutar. Lo importante está dentro del bucle for que es el que hace tratamiento por cada posición de la flag, que por el código se puede interpretar que es el segundo parámetro (un puntero y no un entero cómo interpreta ghidra).

Su equivalente a C real sería:

```
for (int i = 0; i < 0x2d; i++) {
  flag[i] = flag[i] + '0';
}
```

Ahora necesitamos saber qué valores se le pasan por parámetros para poder decodificar la flag. Si pulsamos a referencias podemos ir a la línea que la llama en la función *enjoy_flipper_app*:





```
45      unaff_r8 = malloc(0x41);
46      state_init();
47      iVar2 = init_mutex(auStack92,unaff_r8,0x41);
48      if (iVar2 != 0) {
49        uVar3 = view_port_alloc();
50        view_port_draw_callback_set(uVar3,render_callback,auStack92);
51        view_port_input_callback_set(uVar3,input_callback,iVar1);
52        uVar4 = furi_record_open(&DAT_000102c4);
53        gui_add_view_port(uVar4,uVar3,4);
54        while( true ) {
55          iVar2 = furi_message_queue_get(iVar1,local_68,0x19);
56          uVar5 = acquire_mutex(auStack92,0xffffffff);
57          if ((((iVar2 == 0) && (local_68[0] == '\x01')) && (local_5f == '\0')) &&
58              (local_60 == '\x05')) break;
59          decode_flag(uVar5,&local_50);
60          view_port_update(uVar3);
61          release_mutex(auStack92,uVar5);
```

Sólo necesitamos encontrar el valor que tiene *local_50*:

```
28    local_50 = 0x37313c36;
29    uStack76 = 0x443e4518;
30    uStack72 = 0x4b434235;
31    uStack68 = 0x31020932;
32    local_40 = 0x5000908;
33    uStack60 = 0x33320032;
34    uStack56 = 0x50908;
35    uStack52 = 0x33363433;
36    local_30 = 0x33080532;
37    uStack44 = 0x35340904;
38    uStack40 = 0x5090105;
39    local_24 = 0x4d;
40    iVar1 = furi_message_queue_alloc(1,0xc);
```

Podemos ver que hay una gran cantidad de bytes hardcodeados con valores parecidos entre ellos, esto Ghidra lo suele hacer cuando se trata de un **string** y lo interpreta erróneamente como enteros. Podemos copiar todos esos valores y pasarlos por el *for* que hemos creado antes. Se puede usar un compilador online por comodidad:

main.c

```
#include <stdio.h>

int main()
{
    char flag[] = "\x37\x31\x3c\x36\x44\x3e\x45\x18\x4b\x43\x42\x35\x31\x02\x09\x32\x05" \
                  "\x00\x09\x08\x33\x32\x00\x32\x05\x09\x08\x33\x36\x34\x33\x33\x08\x05" \
                  "\x32\x35\x34\x09\x04\x05\x09\x01\x05\x4d";

    for (int i = 0; i < 0x2d; i++) {
      flag[i] = flag[i] + '0';
    }

    printf("%s\n", flag);
    return 0;
}

```

Y en la salida tenemos lo siguiente:

```
main.c: In function 'main':
main.c:5:90: warning: backslash and newline separated by space
    5 |     char flag[] = "\x37\x31\x3c\x36\x44\x3e\x45\x18\x4b\x43\x42\x35\x31\x02\x09\x32\x05" \
      |
galftnuH{srea29b5098cb0b598cfdcc85bed945915}0U
```

Ha salido desordenado (**galftnuH{sre** es ***flagHunters{***), esto es por culpa de ghidra, ya que al tratarse de un binario en little endian e interpretar las posiciones de memoria como enteros, se cree que están los bytes ordenados como little endian y les da la vuelta para ordenarlos de 4 en 4 (tamaño de una palabra en 32 bits).

Por lo tanto hay que dar un último paso, tener los bytes en el orden correcto. Y también asegurarnos de que estén bien los bytes, con este código ya obtenemos la flag:

main.c

```c
#include <stdio.h>

int main()
{
    char flag[] = "\x37\x31\x3c\x36" \
                  "\x44\x3e\x45\x18" \
                  "\x4b\x43\x42\x35" \
                  "\x31\x02\x09\x32" \
                  "\x05\x00\x09\x08" \
                  "\x33\x32\x00\x32" \
                  "\x00\x05\x09\x08" \
                  "\x33\x36\x34\x33" \
                  "\x33\x08\x05\x32" \
                  "\x35\x34\x09\x04" \
                  "\x05\x09\x01\x05" \
                  "\x00\x00\x00\x4d";

    for (int i = 0; i < sizeof(flag); i++) {
      flag[i] = flag[i] + '0';
    }

    // imprimir con el orden correcto
    for (int i = 3; i <= sizeof(flag); i += 4) {
        for (int j = i; j > (i - 4); j--)
            printf("%c", flag[j]);
    }
    return 0;
}
```

Y obtenemos:

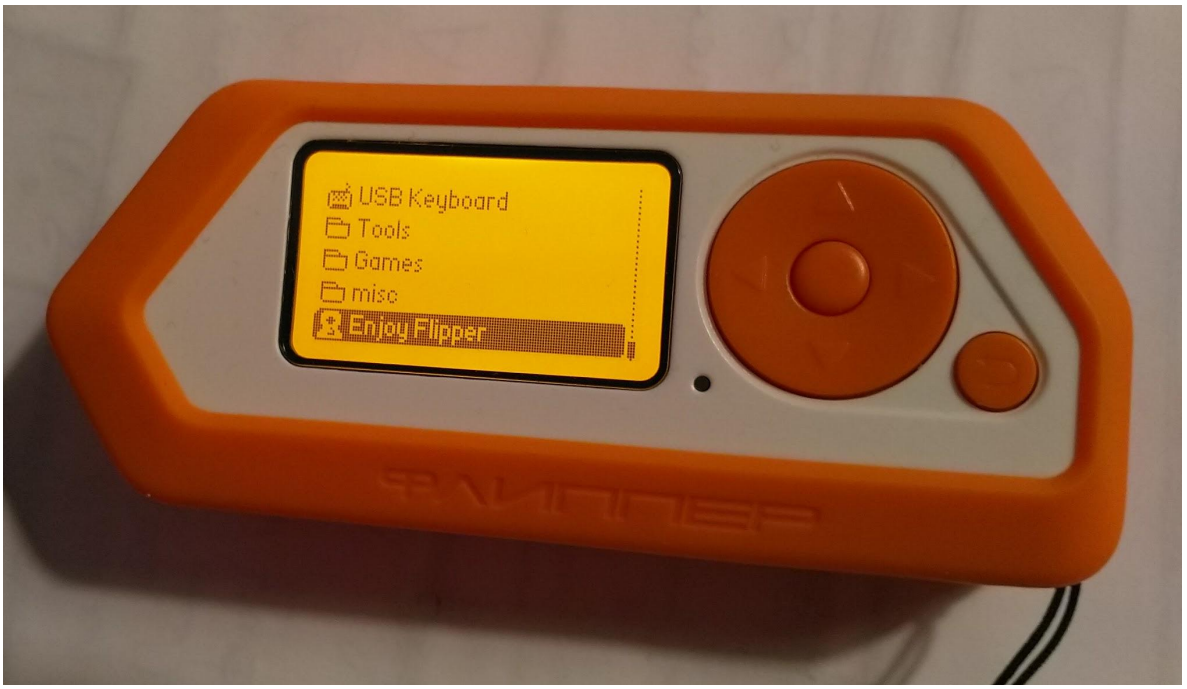
```
flagHunters{b92a8905b0bc8950cdfcb58c49de5195}000
```

Los 3 últimos 0 se pueden considerar padding y descartarse.

2. Contexto

El binario se trata de una aplicación creada para el Flipper Zero que está tan de moda ahora. Se puede cargar en la tarjeta microSD y ejecutarlo como una app externa. En su repositorio de GitHub podemos ver un poco de contexto: https://github.com/flipperdevices/flipperzero-firmware/blob/dev/documentation/AppsOnSDCard.md

Aquí una captura de cómo se muestra la app en el dispositivo:



Al abrirla decodifica y muestra la flag, aunque sale cortada, obligando a hacer el análisis estático del binario (para que no se pueda resolver de esta forma):


flagHunters{b92a8905b0bc