F.L.A.R.E. ON 2016 Challenge

Solution for 1st challenge

Author: Hugo RIFFLET (@l3m0ntr33)

Instruction

As for all step of the FLARE CTF we need to find a flag wich looks like an email address ends with @flare-on.com.

For this 1st challenge we got this file:

```
remnux@remnux:~/Downloads/flare-on$ md5sum challenge1.exe
2caaa4aa5923d026b17d7b38ee410918  challenge1.exe
```

Running the program

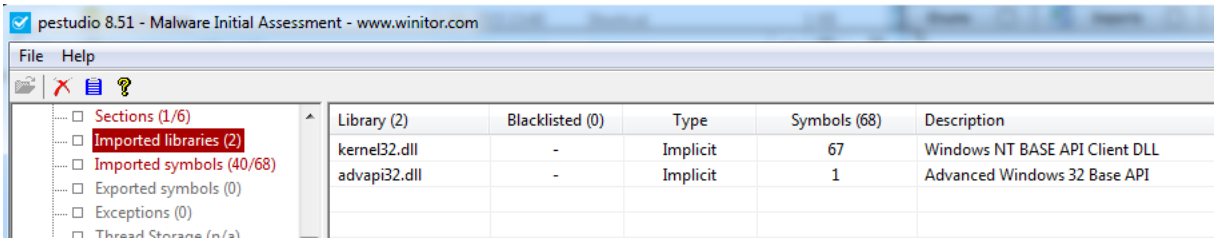
```
c:\samples>challenge1.exe
Enter password:
 multipass
Wrong password

c:\samples>
```

It seems to be a cracking challenge.

File analysis

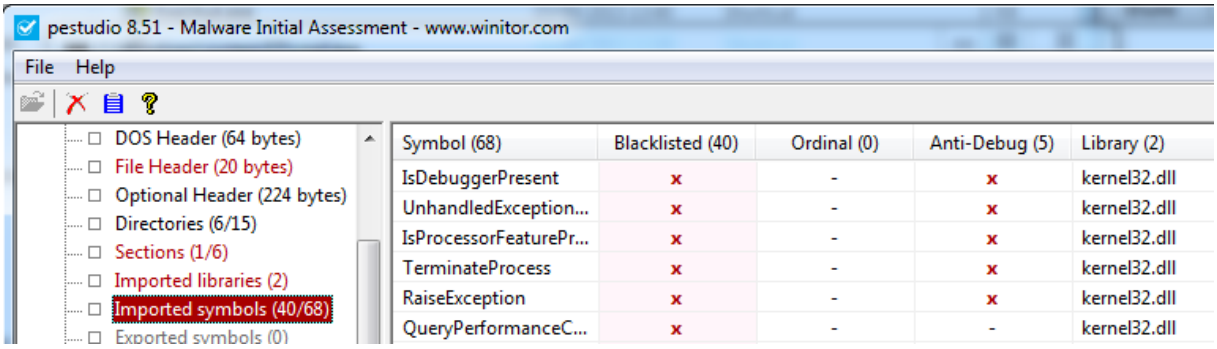
Opening the file with PE Studio.

Library imports:



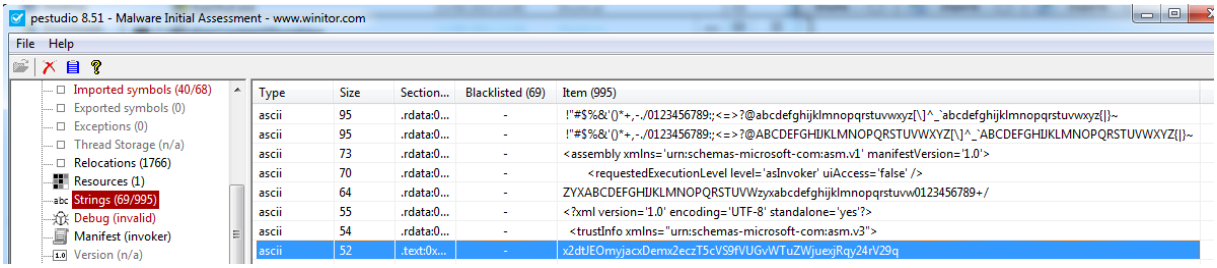
Nothing interesting.

Imported symbols:



Some possible anti-debug symbols imported we will take care of that. The others symbols imported are note very interesting for the moment without a better view of the program functionalities.

Looking at the strings:



The first two strings may be interesting, looks like dictionary or strings used by keyloggers.

Type	Size	Section...	Blacklisted (69)	Item (995)
ascii	95	.rdata:0...	-	!"#$%&'()*+,-./0123456789:;<=>?@abcdefghijklmnopqrstuvwxyz[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
ascii	95	.rdata:0...	-	!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`ABCDEFGHIJKLMNOPQRSTUVWXYZ{|}~

There is also this string that looks like a hash or a password.

ascii	52	.text:0x...	-	x2dtJEOmyjacxDemx2eczT5cVS9fVUGvWTuZWjuexjRqy24rV29q

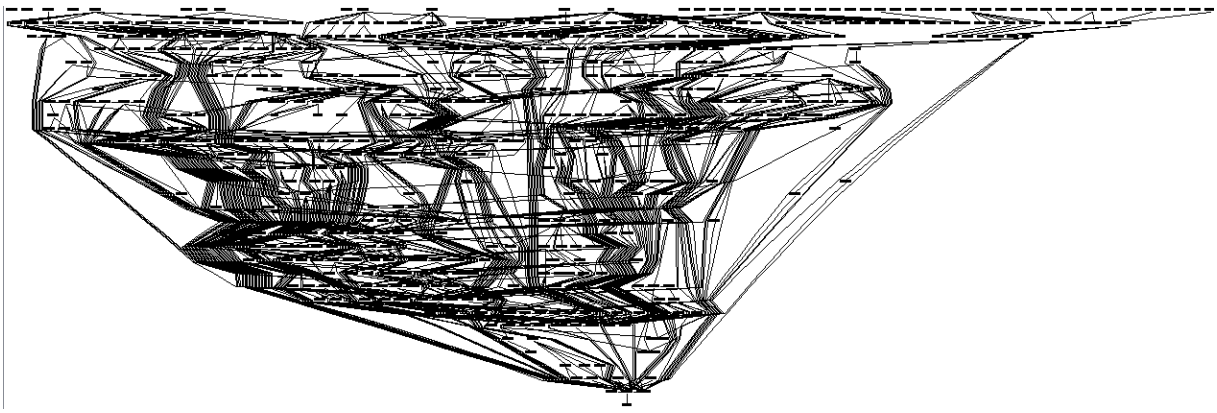
We also find these strings:

Type	Size	Section:Offset	Blacklisted (69)	Item (995)
ascii	15	.text:0xBF95	-	Enter password:
ascii	8	.text:0xBFA8	-	Correct!
ascii	14	.text:0xBFB5	x	Wrong password

It really looks like a cracking challenge.

<u>Disassembling overview</u>

Global call flow:



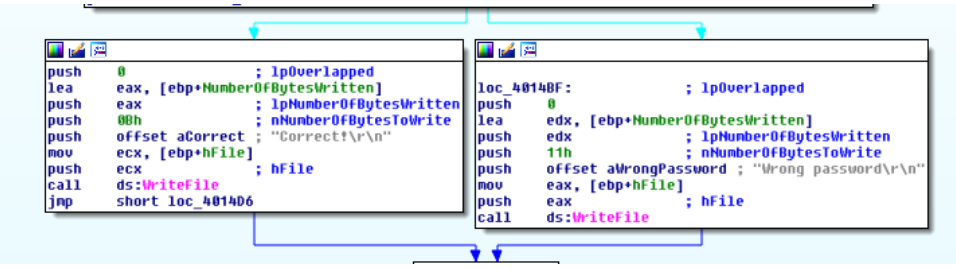
Ouch seems quite complex for a simple password challenge!

Let’s take a look on xref to interesting strings:

```
.rdata:0040D160 aX2dtjeomyjacxd db 'x2dtJEOmyjacxDemx2eczT5cVS9fVUGvWTuZWjuexjRqy24rV29q',0
.rdata:0040D160                                         ; DATA XREF: sub_401420+1F↑o
.rdata:0040D195                 align 4
.rdata:0040D198 aEnterPassword  db 'Enter password:',0Dh,0Ah,0 ; DATA XREF: sub_401420+2E↑o
.rdata:0040D1AA                 align 4
.rdata:0040D1AC aCorrect        db 'Correct!',0Dh,0Ah,0 ; DATA XREF: sub_401420+8E↑o
.rdata:0040D1B7                 align 4
.rdata:0040D1B8 aWrongPassword  db 'Wrong password',0Dh,0Ah,0 ; DATA XREF: sub_401420+A7↑o
```

They all link to the same function sub_401420.

```
; Attributes: bp-based frame

sub_401420 proc near

Buffer= byte ptr -94h
var_14= dword ptr -14h
var_10= dword ptr -10h
var_C= dword ptr -0Ch
hFile= dword ptr -8
NumberOfBytesWritten= dword ptr -4

push    ebp
mov     ebp, esp
sub     esp, 94h
push    0FFFFFFF5h      ; nStdHandle
call    ds:GetStdHandle
mov     [ebp+hFile], eax
push    0FFFFFFF6h      ; nStdHandle
call    ds:GetStdHandle
mov     [ebp+var_C], eax
mov     [ebp+var_10], offset aX2dtjeomyjacxd ; "x2dtJEOmyjacxDemx2eczT5cVS9fVUGvWTuZWju"...
push    0               ; lpOverlapped
lea     eax, [ebp+NumberOfBytesWritten]
push    eax             ; lpNumberOfBytesWritten
push    12h             ; nNumberOfBytesToWrite
push    offset aEnterPassword ; "Enter password:\r\n"
mov     ecx, [ebp+hFile]
push    ecx             ; hFile
call    ds:WriteFile
push    0               ; lpOverlapped
lea     edx, [ebp+NumberOfBytesWritten]
push    edx             ; lpNumberOfBytesRead
push    80h             ; nNumberOfBytesToRead
lea     eax, [ebp+Buffer]
push    eax             ; lpBuffer
mov     ecx, [ebp+var_C]
push    ecx             ; hFile
call    ds:ReadFile
mov     edx, [ebp+NumberOfBytesWritten]
sub     edx, 2
push    edx
lea     eax, [ebp+Buffer]
push    eax
call    sub_401260
add     esp, 8
mov     [ebp+var_14], eax
mov     ecx, [ebp+var_10]
push    ecx
mov     edx, [ebp+var_14]
push    edx
call    sub_402C30
add     esp, 8
test    eax, eax
jnz     short loc_4014BF
```



Ok, quite simple in fact, password is read from user input then its pass to the function sub_401260 and the result returns by sub_401260 (var_14) is passed to sub_402C30 with the string “x2dtJEO…”.

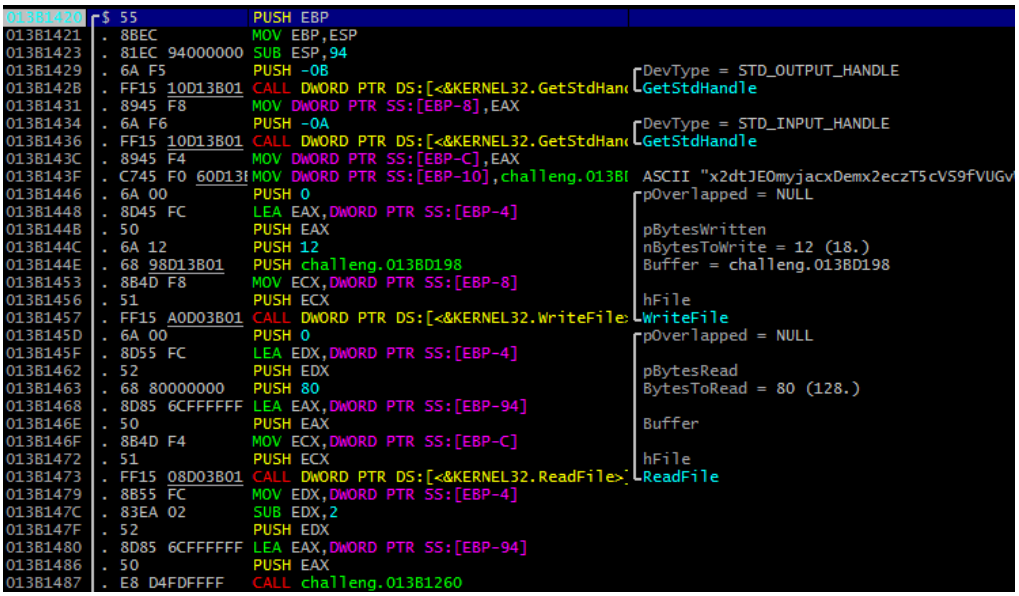
Depending on the result of sub_402C30 “Correct” or “Wrong password” is print on screen.

➔ The string “x2dtJEO…” seems to be the expected result of the call to function sub_401260 with the right password.

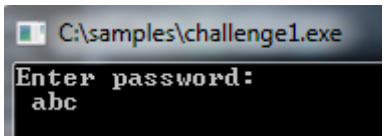
Before going further on the static analysis of sub_401260 we will debug the program to confirm the call flow from the beginning of start point.

Debugging

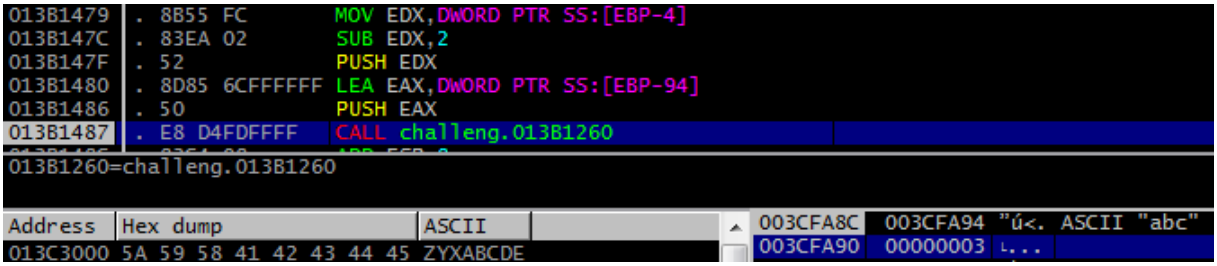
Running the program with Immunity Debugger. Putting a breakpoint at beginning of the sub_401420.



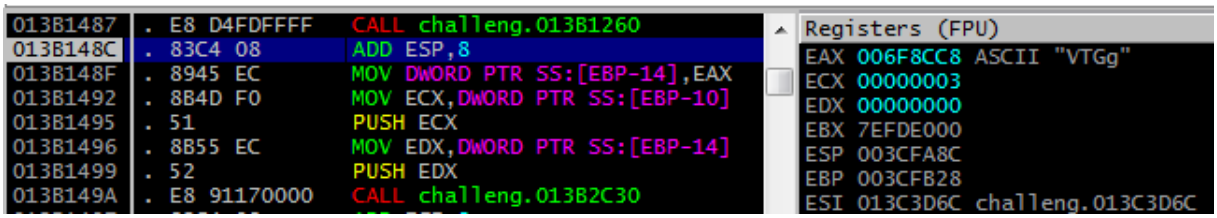
No anti-debugging features seems to be active. Let's try giving "abc" on input.



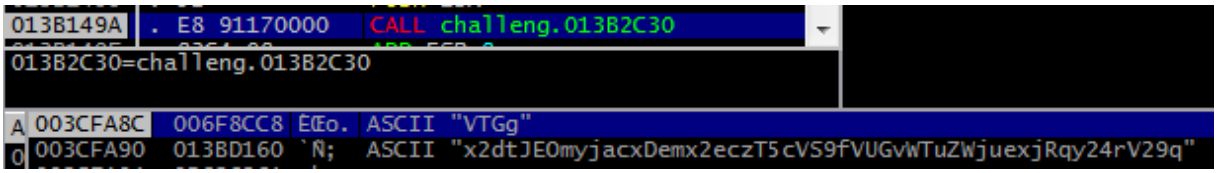
Sub_401260 is called with the password entered and is length (3).



Just step over the sub. A resulting string pointer is put in EAX ("VTGg").



Then the resulting string is passed with the string "x2dt…" to sub_402C30.



Result of sub_402C30 is pushed in EAX, in this case (0xFFFFFFFF). In this case jump is taken to 4014BF printing “Wrong password” on console.

OK so we need to understand the sub_401260 to understand how the result is calculated to found the corresponding password i.e. the flag for the challenge.

Analyzing sub_401260 with IDA

```
push    ebp
mov     ebp, esp
sub     esp, 30h
mov     eax, [ebp+arg_4] ; on move le nombre de char
add     eax, 2           ; on ajoute 2 au nombre de char
xor     edx, edx
mov     ecx, 3
div     ecx              ; on divise le nombre de char+2 par 3
lea     edx, ds:1[eax*4] ; on prend le nombre obtenu * 4 + 1 -> correspond à la longueur totale du résultat soit 9 pour 6 char
mov     [ebp+var_18], edx
mov     eax, [ebp+var_18]
push    eax              ; on pousse ce nombre dans EAX
call    heap_alloc       ; on alloue la mémoire qui servira à stocker le résultat
add     esp, 4
mov     [ebp+var_C], eax
cmp     [ebp+var_C], 0
jnz     short loc_401298
```

At the beginning of the sub we can see that the length of input password is the compute to get the length of result. The result is always a multiple of 4 and a modulo 3. For a password length of 1 to 3 the result length is for 4. For a password length of 4 to 6 the result length is 8. Etc…

So we can already found that the right password has a maximum length of 52/4*3=39 and a minimal length of 37.

'x2dtJEOmyjacxDemx2eczT5cVS9fVUGvWTuZWjuexjRqy24rV29q'

Before **loc_401333**, the three first char are moved in some vars :

1st char → var_28 and var_1C

2nd char → var_20 and var_2C

3rd char → var_24 and var_30

Beginning of loc_401333 :

```
loc_401333:             ; on met le 3ème char dans ECX
mov     ecx, [ebp+var_24]
mov     [ebp+var_30], ecx ; on met le 3ème char dans EBP-30
mov     edx, [ebp+var_28] ; on met le 1er char dans EDX
shl     edx, 10h          ; shl 10h sur le 1er char
mov     eax, [ebp+var_2C] ; 2ème char dans EAX
shl     eax, 8            ; shl 8h sur le 2ème char
add     edx, [ebp+var_30] ; on ajoute le 3ème char à EDX
add     eax, edx
mov     [ebp+var_10], eax ; contient les 3 char décalé d'1 octet chacun 00 61 62 63
```

Assuming that our password is “abc”:

- a → 61h → 0000 0000 0000 0000 0110 0001
- b → 62h → 0000 0000 0000 0000 0110 0010
- c → 62h → 0000 0000 0000 0000 0110 0011

After these first set of instructions each char is shift on the 32 bit word and we got this in **var_10**:

0	a	b	c
0000 0000	0110 0001	0110 0010	0110 0011

Then we got these operations:

```
mov     ecx, [ebp+var_10]
shr     ecx, 12h
and     ecx, 3Fh        ; ECX contient le résultat de nos combinaisons
mov     edx, [ebp+var_C]
add     edx, [ebp+var_8] ;  on ajoute 0
mov     al, byte_413000[ecx] ; on récupère le caractère en utilisant ECX + 1103000
mov     [edx], al       ; on copie le résultat dans la mémoire réservée
```

We apply an “shr 12h” on var_10 and we keep only the last 6 bits with “and 3Fh”.

“Shr 12h” and “and 3Fh”

0	0	0	18h
0000 0000	0000 0000	0000 0000	0001 1000

➔ **We got the 6 higher bits of 1st char.** This value is used to extract a char from a static dictionary stored at 313000 in our case. 31300 + 18 = 18h corresponding to ‘**V**’ char.

```
Address   Hex dump                ASCII
00313000  5A 59 58 41 42 43 44 45  ZYXABCDE
00313008  46 47 48 49 4A 4B 4C 4D  FGHIJKLM
00313010  4E 4F 50 51 52 53 54 55  NOPQRSTU
00313018  56 57 7A 79 78 61 62 63  VWzyxabc
00313020  64 65 66 67 68 69 6A 6B  defghijk
00313028  6C 6D 6E 6F 70 71 72 73  lmnopqrs
00313030  74 75 76 77 30 31 32 33  tuvw0123
00313038  34 35 36 37 38 39 2B 2F  456789+/
```

The same kind of operation is repeated 3 times more with different SHR instructions.

⇨ *Second SHR*

```
shr     edx, 0Ch
and     edx, 3Fh
```

“Shr 0Ch” and “with “and 3Fh”

0	0	0	16h
0000 0000	0000 0000	0000 0000	0001 0110

➔ **We got the 2 last bits of 1st char and 4 higher bits of 2nd char.** This value is used to extract a char from a static dictionary stored at 313000 in our case. 31300 + 16 = 16h corresponding to ‘**T**’ char.

⇨ *Third SHR*

```
shr     eax, 6
and     eax, 3Fh
```

"Shr 06h" and "with "and 3Fh"

0	0	0	09h
0000 0000	0000 0000	0000 0000	0000 1001

➔ **We got the 4 last bits of 2nd char and 1 higher bits of 3rd char.** This value is used to extract a char from a static dictionary stored at 313000 in our case. 31300 + 09 = 09h corresponding to '**G**' char.

⇨ *Fourth SHR*

```
shr     ecx, 0
and     ecx, 3Fh
```

"Shr 00h" and "with "and 3Fh"

0	0	0	23h
0000 0000	0000 0000	0000 0000	0010 0011

➔ **We got the 6 last bits of 4th char.** This value is used to extract a char from a static dictionary stored at 313000 in our case. 31300 + 09 = 09h corresponding to '**g**' char.

Synthesis

So we could see that the resulting string is made of shr operand used to select 4 times 6 bits of each original password char and then use a dictionary to make them match a specific char.

Knowing that we can now extract from the hardcoded string the flag:
"x2dtJEOmyjacxDemx2eczT5cVS9fVUGvWTuZWjuexjRqy24rV29q".

Python solving script

I just make a very simple python script using the challenge dictionary to retrieve the flag. You could find at the end of the document the complete script.

```
remnux@remnux:~$ ./flare1.py
########## DECODE #########
# Hash :  x2dtJEOmyjacxDemx2eczT5cVS9fVUGvWTuZWjuexjRqy24rV29q
############################
Password :  sh00ting_phish_in_a_barrel@flare-on.com
########## ENCODE #########
# Password :  sh00ting_phish_in_a_barrel@flare-on.com
############################
Pass encoded :  x2dtJEOmyjacxDemx2eczT5cVS9fVUGvWTuZWjuexjRqy24rV29q
remnux@remnux:~$
```

Validating the challenge:

```
c:\samples>challenge1.exe
Enter password:
 sh00ting_phish_in_a_barrel@flare-on.com
Correct!

c:\samples>
```

Python 3 script

```
#!/usr/bin/python3
import binascii

myhash="x2dtJEOmyjacxDemx2eczT5cVS9fVUGvWTuZWjuexjRqy24rV29q"
myDico="ZYXABCDEFGHIJKLMNOPQRSTUVWzyxabcdefghijklmnopqrstuvw0123456789+/"


def decode(passenc, dico):
      index = 1
      password=""
      print("########## DECODE #########")
      print("# Hash : ", passenc)
      print("############################")
      #print(password)
      for char in passenc:
            if index == 1:
                  char1 = 0
                  char2 = 0
                  char3 = 0
                  #print("###### INDEX ",index,"######")
                  #print("compute: ",char)
                  tabid = dico.find(char)
                  #print(char," found at index ", tabid, " soit ",
bin(tabid))
                  tmp = tabid << 2
                  #print("char1 avant: ", bin(char1))
                  char1 = char1 | tmp
                  #print("char1 apres: ", bin(char1))
            if index == 2:
                  #print("###### INDEX ",index,"######")
                  #print("compute: ",char)
                  tabid = dico.find(char)
                  #print(char," found at index ", tabid, " soit ",
bin(tabid))
                  tmp = tabid & int('0000000000110000',2)
                  tmp = tmp >> 4
                  #print("char1 avant: ", bin(char1))
```

```
                char1 = char1 + tmp
                #print("char1 apres: ", bin(char1)," soit ", chr(char1))
                password = password + chr(char1)


                tmp = tabid & int('00001111',2)
                tmp2 = tmp << 4
                #print("char2 avant: ", bin(char2))
                char2 = char2 | tmp2
                char2 = char2 & int('0000000011111111',2)
                #print("char2 apres: ", bin(char2))
                #print(char2)

        if index == 3:
                #print("###### INDEX ",index,"######")
                #print("compute: ",char)
                tabid = dico.find(char)
                #print(char," found at index ", tabid, " soit ",
bin(tabid))

                #on recupere les 4 bits de poids faible de char2
          #on fait un AND 0011 1100
                tmp = tabid & int('00111100',2)
                tmp = tmp >> 2
                #print("char2 avant: ", bin(char2))
          #on fait un ADD sur char2
                char2 = char2 + tmp
                #print("char2 apres: ", bin(char2)," soit ", chr(char2))
                password = password + chr(char2)

          #on recupere les 2 bits de poid fort de char3
          #on fait un AND 00000011
                tmp = tabid & int('00000011',2)
          #on décale de 6 vers la gauche
                tmp2 = tmp << 6
                #print("char3 avant: ", bin(char3))
                    #on ajoute a char3
                char3 = char3 | tmp2
                char3 = char3 & int('0000000011111111',2)
                #print("char3 apres: ", bin(char3))
                    #print(char2)

        if index == 4:
                #print("###### INDEX ",index,"######")
                #print("compute: ",char)
                tabid = dico.find(char)
```

```
                #print(char," found at index ", tabid, " soit ",
bin(tabid))

                    #on recupere les 6 bits de poids faible de char3
                    #on fait un AND 0011 1111
                tmp = tabid & int('00111111',2)
                #print("char3 avant: ", bin(char3))
                    #on fait un ADD sur char3
                char3 = char3 + tmp
                #print("char3 apres: ", bin(char3)," soit ", chr(char3))
                password = password + chr(char3)
                index = 0
          index += 1
    print("Password : ", password)

def encode(password, dico):
    index = 0
    char1 = 0
    char2 = 0
    char3 = 0
    char4 = 0
    passenc=""
    print("########## ENCODE #########")
    print("# Password : ", password)
    print("############################")
    for char in password:
          if (index % 3) == 0:
                tmphash = 0
                char1 = int.from_bytes(char.encode('utf-8'),
byteorder='big')
                tmphash = char1 << 16
                #print("Char1 = ", bin(char1))
                #print("tmphash = ", bin(tmphash))
          if (index % 3)  == 1:
                char2 = int.from_bytes(char.encode('utf-8'),
byteorder='big')
                tmphash = tmphash | (char2 << 8)
                #print("Char2 = ", bin(char2))
                #print("tmphash = ", bin(tmphash))
          if (index % 3)  == 2:
                char3 = int.from_bytes(char.encode('utf-8'),
byteorder='big')
                tmphash = tmphash | char3
                #print("Char3 = ", bin(char3))
                #print("tmphash = ", bin(tmphash))
                #On peut maintenant obtenir les caracteres du hash
```

```
                char1 = (tmphash >> 18) &
int('00000000000000000000000000111111',2)
                #print("Char 1 = ", dico[char1])
                char2 = (tmphash >> 12) &
int('00000000000000000000000000111111',2)
                #print("Char 2 = ", dico[char2])
                char3 = (tmphash >> 6) &
int('00000000000000000000000000111111',2)
                #print("Char 3 = ", dico[char3])
                char4 = (tmphash >> 0) &
int('00000000000000000000000000111111',2)
                #print("Char 4 = ", dico[char4])
                passenc = passenc + dico[char1] + dico[char2] +
dico[char3] + dico[char4]
            index += 1
      print("Pass encoded : ", passenc)


decode(myhash,myDico)
encode("sh00ting_phish_in_a_barrel@flare-on.com",myDico)
```