ExaTrack

Solution du challenge « re_crypto » #NDH16

ExaTrack - Stéfan Le Berre (stefan.le-berre [at] exatrack.com)

Il y a quelques jours s'est déroulé la Nuit Du Hack, 16ème édition. Lors de cet événement des challenges sont mis à disposition du public, chaque épreuve permet de gagner des points, et celui qui a le plus de points à la fin de l'événement gagne des lots.

Cette année j'ai décidé de développer une épreuve de reverse engineering / crypto pour l'événement. Malheureusement je crois que personne ne l'a résolue, je tiens donc à m'excuser après des challengeurs si mon épreuve a été un peu trop optimiste sur la solvabilité dans le temps imparti. Je vais présenter ici comment j'ai résolu cette épreuve (plusieurs solutions sont possibles).

Contexte

Deux fichiers sont fournis aux challengeurs :

- un exécutable qui chiffre un fichier avec un mot de passe entré par l'utilisateur ;
- un document texte chiffré.

L'objectif est donc de déchiffrer le document sans pour autant avoir l'outil de déchiffrement ni le mot de passe.

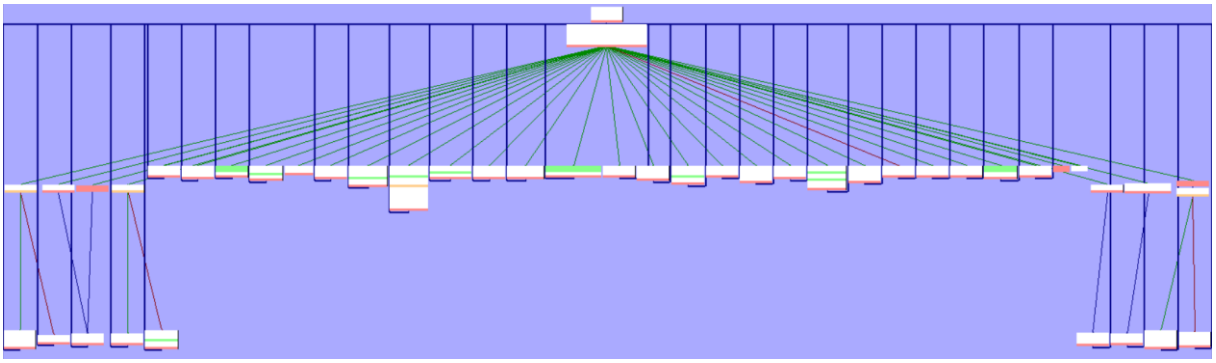
Analyse de l'exécutable

Désobfuscation

En ouvrant le binaire on voit assez rapidement que le code est obfusqué. Un système de `push`/`ret` permet d'effectuer un jmp sur l'adresse choisie.

```
push rbp
push rbx
sub rsp, 88h
push 3fh
push 401530h
ret

401530h:
pushfq
push rax
mov rax, [rsp+10h]
shl rax, 3
add rax, 405000h
mov rax, [rax]
mov [rsp+10h], rax
pop rax
popfq
ret
```

On remarque qu'un argument est empilé (ici 0x3f) et qu'il est utilisé comme index dans le tableau à l'adresse 0x45000. La valeur obtenue est écrite à la place de l'index empilé, puis un `ret` permet de sauter à cette adresse. Nous sommes face à un cas de flattening assez simple, on le voit bien si on essaie de relier le graph par de l'exécution symbolique, Metasm le fait très bien :

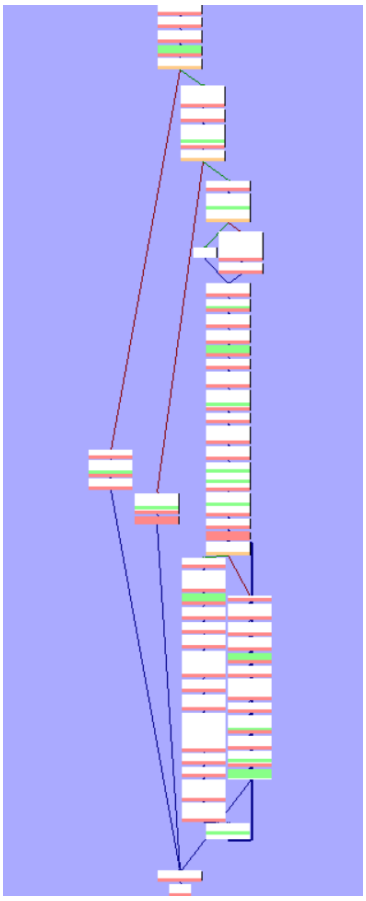


Nous allons donc développer un petit script qui remplacera la séquence d'instruction par un `jmp` pour permettre aux désassembleurs de suivre plus facilement ce qui se passe.

```
# encoding: ASCII-8BIT
require './metasm/metasm'
include Metasm

target = ARGV.shift

Encoding.default_internal = Encoding.find('ASCII-8BIT')
Encoding.default_external = Encoding.find('ASCII-8BIT')

raw_datas = open(target, "rb") {|io| io.read }.force_encoding("ASCII-8BIT")

decodedfile = AutoExe.decode_file(target)
entrypoints =  decodedfile.get_default_entrypoints
dasm = decodedfile.disassembler

puts "  [*] Fast disassemble of binary..."
dasm.disassemble_fast_deep(*entrypoints)

base_table = 0x405000
jmp_pattern = "\x68\x30\x15\x40\x00\xc3" # push 0x401530;ret

offset = raw_datas.index(jmp_pattern)
# pp raw_datas
while offset != nil
    puts "Pattern at %x" % (dasm.fileoff_to_addr(offset))
    if raw_datas[offset-2] == "\x6a" # short push
        index_va = dasm.fileoff_to_addr(offset-2)
        di = dasm.disassemble_instruction(index_va)
        index_addr = dasm.normalize(di.instruction.args[0])
        # compute target address by ID
        target_addr = dasm.decode_dword(base_table+(index_addr*8))
        sc = Shellcode.assemble(AMD64.new, "jmp 0x#{target_addr.to_s(16)}")
        sc.base_addr = di.address
        raw = sc.encode().encoded.data
        raw_datas[offset-2,raw.length] = raw
    end
```

```
    offset = raw_datas.index(jmp_pattern, offset+jmp_pattern.length)
end

open("desobfu_#{target}","wb"){|io| io.write(raw_datas)}
puts "Desobfuscated file : desobfu_#{target}"
```

Nous obtenons ainsi des graphiques comme suit :



A ce moment les différents décompilateurs peuvent travailler efficacement sur les instructions pour en sortir un code C à peu près valide.

Analyse des opérations cryptographiques

La première chose à faire quand on commence un challenge est de débugger l'exécutable pour identifier où nos entrées sont utilisées. On peut rapidement s'apercevoir qu'un hash est généré depuis le mot de passe. Ce hash sera ensuite utilisé dans les opérations cryptographiques.

En continuant avec le décompilateur de Metasm nous pouvons récupérer un code C exploitable :

```
HeapAlloc();
var_38 = *iat_HeapAlloc;
loc_403550h();
rsp[4] = 0;
ReadFile();
CloseHandle();
loc_401db6h();
var_20 = *(arg_8_a0 + 1);
var_28 = 0xA247A1E304738C7B;
var_18 = 0;
while (var_18 < var_14) {
        loc_4017bch();
        rax_a29 = var_38;
        rax_a33 = var_18;
        *(__int64*)((rax_a33 | (-0x100000000 * (rax_a33 < 0))) + var_38) = (*(__int64*)(var_38 + (var_18 | (-0x100000000 * (var_18 < 0)))) ^ var_28);
        rax_a36 = var_38;
        *(int*)((var_18 | (0xFFFFFFFF00000000 * (var_18 < 0))) + 8 + var_38) = (*(__int64*)(rax_a36 + rax_a36 + 8) ^ var_28);
        var_28 = (*(__int64*)(rax_a29 + rax_a29 + 8) ^ *(__int64*)(var_38 + (var_18 | (-0x100000000 * (var_18 < 0)))));
        loc_401666h();
        var_20 = var_20;
        var_18 += 16;
}
```

Ici nous voyons clairement une boucle qui prendra des blocs de 16 octets (`var_18 += 16;`) pour appliquer des transformations.

A chaque tour de chiffrement nous voyons deux fonctions appelées, `0x4017bc` au début, puis `0x401666` à la fin.

Commençons par la fonction `0x401666`, elle appelle deux fonctions. Ces deux fonctions effectuent beaucoup d'opérations logiques, les inverser peut être fastidieux, nous allons donc demander à Metasm d'effectuer un backtrace du registre de retour pour avoir une synthèse des opérations.

Pour la fonction `0x401ce5` nous aurons le résultat :

```
rax|rdx
((rax&0ffffffffffffffffh)>>(rcx&3fh))|rdx
((qword ptr [rbp+10h]&0ffffffffffffffffh)>>(rcx&3fh))|rdx
((qword ptr [rbp+10h]&0ffffffffffffffffh)>>(rax&3fh))|rdx
((qword ptr [rbp+10h]&0ffffffffffffffffh)>>(rcx&3fh))|rdx
((qword ptr [rbp+10h]&0ffffffffffffffffh)>>((rcx-(rax&0ffffffffh))&3fh))|rd
((qword ptr [rbp+10h]&0ffffffffffffffffh)>>((-(rax&0ffffffffh)+40h)&3fh))|r
((qword ptr [rbp+10h]&0ffffffffffffffffh)>>((-(qword ptr [rbp+18h]&0fffffff
((qword ptr [rbp+10h]&0ffffffffffffffffh)>>((-(qword ptr [rbp+18h]&0fffffff
((qword ptr [rbp+10h]&0ffffffffffffffffh)>>((-(qword ptr [rbp+18h]&0fffffff
((qword ptr [rbp+10h]&0ffffffffffffffffh)>>((-(qword ptr [rbp+18h]&0fffffff
((qword ptr [rbp+10h]&0ffffffffffffffffh)>>((-(qword ptr [rbp+18h]&0fffffff
((qword ptr [rbp+10h]&0ffffffffffffffffh)>>((-(qword ptr [rbp+18h]&0fffffff
((qword ptr [rbp+10h]&0ffffffffffffffffh)>>((-(qword ptr [rbp+18h]&0fffffff
((qword ptr [rbp+10h]&0ffffffffffffffffh)>>((-(qword ptr [rbp+18h]&3fh)+40h
((qword ptr [rbp+10h]&0ffffffffffffffffh)>>((-(rdx&3fh)+40h)&3fh))|(qword p
((rcx&0ffffffffffffffffh)>>((-(rdx&3fh)+40h)&3fh))|(rcx<<(rdx&3fh))
```

La dernière ligne est le résultat réduit des opérations précédentes. Les lignes précédentes montrent l'ensemble des contraintes imposées au fur et à mesure de l'analyse des instructions.

Typiquement l'opération réalisée est un « ROL », qui prend comme valeur d'entrée `RCX` et comme nombre de bits à déplacer `RDX`.

La deuxième fonction, elle, effectue un « ROR ». Et l'utilisation des deux est fait comme suit :

```
new_password_hash = __ROR8__(password_hash ^ __ROL8__(password_hash, 3),2);
```

Nous sommes donc dans le cas d'une rotation de clé avec des opérations logiques. Nous pouvons donc dire que la fonction `0x401666`, effectue l'opération ci-dessus.

Nous ne voyons pas d'écriture de données dans cette fonction, nous pouvons donc estimer que le chiffrement est fait dans l'autre fonction, `0x4017bc`. Cette fonction prend deux arguments, le premier est utilisé pour mettre à jour une zone mémoire du binaire (cette zone mémoire contient des données arbitraires propres à l'exécutable). Cette mise à jour inclus une rotation du buffer de 0x10 octets à chaque chiffrement de bloc. Si on analyse rapidement les valeurs initiales du buffer à l'adresse 0x404040. On se rend compte que celui-ci contient toutes les valeurs de 0 à 255 dans un ordre aléatoire.

```
for (i = 0; i < 256; i++){
    ((char *)0x404040)[i] ^= (password_hash&0xff);
}
```

Nous pouvons noter que la mise à jour étant limitée par `(arg_1&0xff)` ne nombre de configurations possibles s'élève à 256.

Puis le deuxième argument pointe tout simplement sur les données du fichier à chiffrer (ceci se voit rapidement avec un débugger). Ce pointeur est utilisé dans la boucle suivante :

```
for (i = 0; i < 16; i++){
    input_datas[i] ^= ((char *)0x404040)[i];
}
```

Nous savons donc que nos données sont chiffrées à partir du tableau à l'adresse `0x404040` qui fait 256 octets.

Enfin, `0x4017bc` appelle la fonction `0x401e79` qui prend en argument notre hash et nos données :

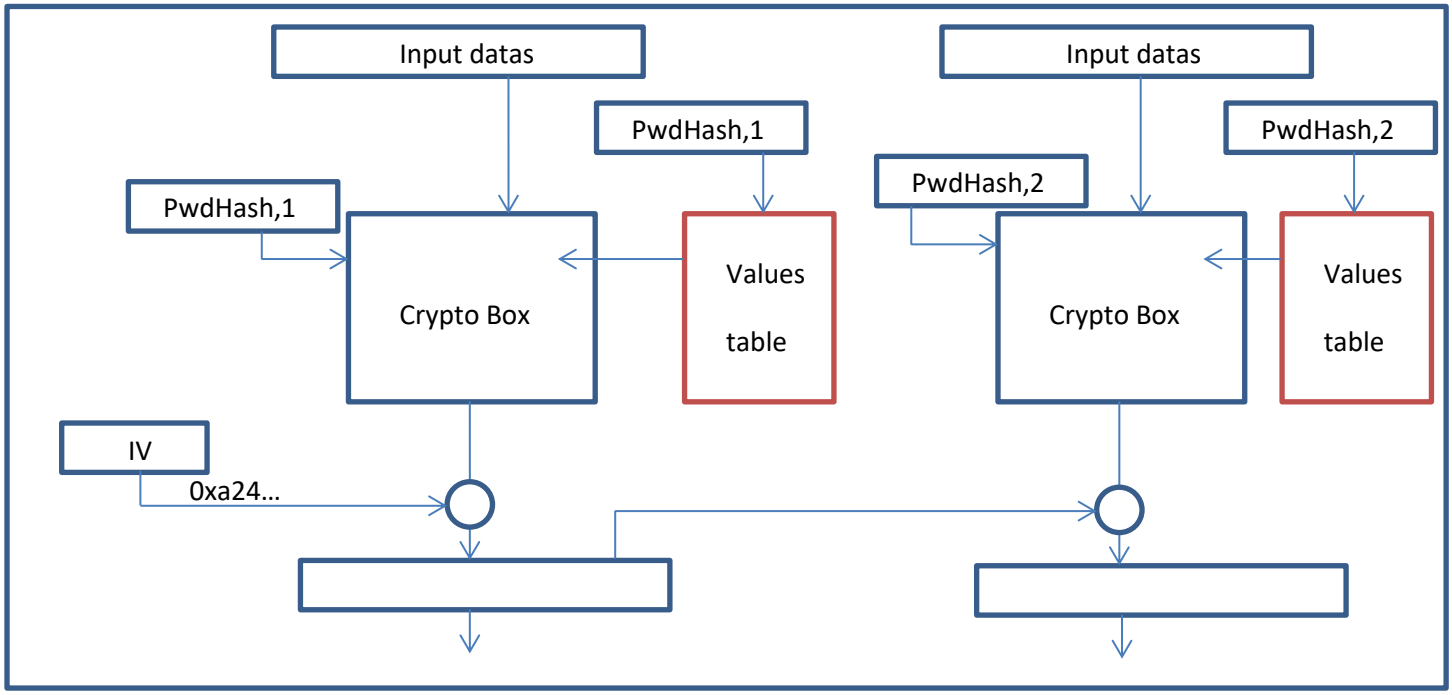
```
QWORD sub_401e79(QWORD password_hash, unsigned char* cypher_text){
    unsigned char tcar;
    int i;
    unsigned int index_1,index_2;

    for (i=0; i < 15; i++){
        index_1 = (password_hash&(0xf<<i))>>(i);
        index_2 = (password_hash&(0xf0<<(i)))>>((i+4));

        tcar = datas[index_1];
        cypher_text[index_1] = cypher_text[index_2];
        cypher_text[index_2] = tcar;
    }

    return 0;
}
```

Cette fonction a pour but d'effectuer des permutations d'octets dans notre bloc suivant le hash du mot de passe.

Revenons à notre boucle traitant les blocs, on peut voir que chaque bloque est xoré en fonction de son précédent, avec un QWORD, le premier étant `0xa247a1e304738c7b`. Ce QWORD xor indépendamment les deux mots de 8 octets de notre bloc courant.

Système de chiffrement et faiblesses

Nous avons à peu près fait le tour des parties importantes à appréhender du challenge. Il nous est maintenant possible de représenter le système de chiffrement :



Pour identifier une faiblesse cryptographique évidente, un bon moyen est de chiffrer un document ne contenant que des 0. Après un certain nombre de blocs nous pouvons observer une répétition :

```
008e0h dfabc0aeb1338328 b73fa3f0099d2316 757531d16bc749b5 03f201e1fc2d7c0a
00900h 5ff34d1b8b711498 fa9d9a77063c8917 839ee83b3518aa24 39e6c92f8d8ac71c
00920h b2ba781d57ea221a e4353949417ffdbc 90fdc6c638d72297 3c20e6e3669639a1
00940h 3b3a47842df5671a 2e7d32ee64a3c0f3 381fe1b3833576d4 2b5280ff28025d73
00960h c3e16f22fa210f04 f9f0c7e24bc8ebdc 9fcb5fe3b498a82e 74ce2df0083edde4
00980h 75b6b3e83645a117 28a87ff68473fc9c 31817cbc7f53dbef ec68cdd7009f15b0
009a0h b0db5a7ae2df927f cdf0999564fa746a 5801bdf03331343a b2c7422bf99ea375
009c0h e00a0703b3d849d6 aac00d7fce6119d7 dcc1e46b3b4dd05e 30024b73c3340003
009e0h dfabc0aeb1338328 b73fa3f0099d2316 757531d16bc749b5 03f201e1fc2d7c0a
00a00h 5ff34d1b8b711498 fa9d9a77063c8917 839ee83b3518aa24 39e6c92f8d8ac71c
00a20h b2ba781d57ea221a e4353949417ffdbc 90fdc6c638d72297 3c20e6e3669639a1
00a40h 3b3a47842df5671a 2e7d32ee64a3c0f3 381fe1b3833576d4 2b5280ff28025d73
00a60h c3e16f22fa210f04 f9f0c7e24bc8ebdc 9fcb5fe3b498a82e 74ce2df0083edde4
00a80h 75b6b3e83645a117 28a87ff68473fc9c 31817cbc7f53dbef ec68cdd7009f15b0
00aa0h b0db5a7ae2df927f cdf0999564fa746a 5801bdf03331343a b2c7422bf99ea375
00ac0h e00a0703b3d849d6 aac00d7fce6119d7 dcc1e46b3b4dd05e 30024b73c3340003
00ae0h dfabc0aeb1338328 b73fa3f0099d2316 757531d16bc749b5 03f201e1fc2d7c0a
00b00h 5ff34d1b8b711498 fa9d9a77063c8917 839ee83b3518aa24 39e6c92f8d8ac71c
00b20h b2ba781d57ea221a e4353949417ffdbc 90fdc6c638d72297 3c20e6e3669639a1
00b40h 3b3a47842df5671a 2e7d32ee64a3c0f3 381fe1b3833576d4 2b5280ff28025d73
00b60h c3e16f22fa210f04 f9f0c7e24bc8ebdc 9fcb5fe3b498a82e 74ce2df0083edde4
00b80h 75b6b3e83645a117 28a87ff68473fc9c 31817cbc7f53dbef ec68cdd7009f15b0
00ba0h b0db5a7ae2df927f cdf0999564fa746a 5801bdf03331343a b2c7422bf99ea375
00bc0h e00a0703b3d849d6 aac00d7fce6119d7 dcc1e46b3b4dd05e 30024b73c3340003
00be0h dfabc0aeb1338328 b73fa3f0099d2316 757531d16bc749b5 03f201e1fc2d7c0a
00c00h 5ff34d1b8b711498 fa9d9a77063c8917 839ee83b3518aa24 39e6c92f8d8ac71c
```

Les patterns se répètent tous les 0x100 octets. A chaque tour notre bloc de 256 octets servant à chiffrer les données effectue justement une rotation complète.

Ceci étant elle est normalement xorée avec un hash de clé. Ce hash de clé, comme nous l'avons vu précédemment, subit les opérations suivantes à chaque tour :

```
new_password_hash = __ROR8__(password_hash ^ __ROL8__(password_hash, 3),2);
```

Seulement cette suite d'opérations n'est pas viable et s'effrite au fil des tours pour finir à 0. Ceci est dû à des bits qui s'annulent régulièrement. Et une fois passé à 0 il n'y a plus de XOR de la table de chiffrement, ni de permutation de nos données une fois chiffré, ce qui simplifie grandement nos calculs.

Nous avons donc une table de chiffrement dans une configuration que nous ne connaissons pas, mais qui est limité à 256 possibilités (lié au dernier `(password_hash&0xff);` effectif).

Développement du système de déchiffrement

Nous allons réimplémenter le processus cryptographique mais nous partirons du dernier bloc, que nous dé-xorerons avec le précédent, puis nous chercherons une configuration dans la table de chiffrement qui nous donne de la donnée texte. Une fois une configuration identifiée nous essayerons de déchiffrer un plus large scope et verrons si les données déchiffrées sont toujours du texte et si elles ont du sens.

Le code de déchiffrement est le suivant :

```
import sys
import struct

datas = open(sys.argv[1],'rb').read()

blocs = []
for i in range(0, len(datas)/16):
    blocs.append(datas[i*16:(i*16)+16])

prev_iv = struct.unpack("Q", blocs[-2][0:8])[0] ^ struct.unpack("Q", blocs[-
2][8:16])[0]
# output of the crypto box of the last bloc, before xoring with the preceding
one
un_iv_block1 = struct.unpack("Q", blocs[-1][0:8])[0] ^ prev_iv
un_iv_block2 = struct.unpack("Q", blocs[-1][8:16])[0] ^ prev_iv


# datas referenced at 0x404040
sbox = [153, 251, 123, 193, 157, 64, 243, 15, 38, 206, 145, 86, 173, 112, 116,
121, 53, 51, 137, 14, 57, 111, 102, 59, 186, 174, 196, 131, 85, 184, 8, 158,
185, 37, 71, 170, 69, 45, 226, 52, 129, 72, 213, 18, 81, 12, 154, 142, 48, 93,
106, 253, 27, 75, 208, 26, 150, 130, 255, 235, 79, 10, 95, 76, 35, 239, 80, 60,
128, 149, 96, 212, 21, 244, 17, 98, 165, 181, 189, 120, 62, 110, 166, 97, 179,
117, 245, 133, 215, 201, 240, 40, 217, 0, 178, 144, 47, 195, 113, 216, 203, 134,
74, 63, 136, 232, 70, 115, 135, 231, 7, 44, 177, 54, 4, 67, 124, 28, 190, 194,
105, 210, 237, 242, 188, 180, 175, 248, 228, 94, 99, 23, 49, 229, 200, 183, 46,
43, 197, 171, 34, 151, 205, 92, 207, 198, 241, 152, 233, 211, 161, 140, 68, 155,
199, 42, 247, 31, 191, 209, 118, 148, 119, 223, 176, 162, 141, 126, 41, 90, 187,
160, 219, 19, 100, 163, 50, 78, 1, 143, 3, 249, 32, 127, 39, 168, 36, 9, 236,
58, 11, 2, 61, 192, 6, 167, 13, 108, 56, 55, 114, 87, 169, 109, 214, 147, 254,
221, 139, 204, 101, 107, 202, 234, 222, 24, 138, 156, 220, 22, 182, 224, 20, 82,
77, 146, 230, 84, 5, 252, 25, 132, 16, 66, 159, 172, 29, 83, 218, 104, 103, 65,
88, 91, 164, 125, 122, 33, 89, 225, 246, 227, 250, 30, 238, 73]

def try_decrpt(box_key, box_offset):
    new_text = ""
    for i in range(0,len(blocs)-1):
        cid = len(blocs)-i-1
        prev_iv = struct.unpack("Q", blocs[cid-1][0:8])[0] ^ struct.unpack("Q",
blocs[cid-1][8:16])[0]
        un_iv_block1 = struct.unpack("Q", blocs[cid][0:8])[0] ^ prev_iv
        un_iv_block2 = struct.unpack("Q", blocs[cid][8:0x10])[0] ^ prev_iv
        qw1, qw2 = struct.unpack("QQ",get_xor_sbox(box_key, box_offset))
        new_text =
struct.pack("Q",qw1^un_iv_block1)+struct.pack("Q",qw2^un_iv_block2)+new_text
        box_offset = box_offset - 1
        if box_offset < 0:
            box_offset = 0xf
    return new_text
```

```
def is_string(test_str):
    for a in test_str:
        if not((0x20 <= ord(a) <= 0x7e) or ord(a) == 0xa or ord(a) == 0x9 or
ord(a) == 0xd or ord(a) == 0x0):
            return False
    return True

def get_xor_sbox(key, index):
    global sbox
    new_bloc = ''.join(chr(a ^ key) for a in sbox[index*0x10:(index*0x10)+16])
    return new_bloc

for ckey in range(0,256):
    coffset = ((len(datas)>>4) % 0x10)
    qw1, qw2 = struct.unpack("QQ",get_xor_sbox(ckey, coffset))
    if is_string(struct.pack("Q",qw1^un_iv_block1)) and
is_string(struct.pack("Q",qw2^un_iv_block2)):
        qw1, qw2 = struct.unpack("QQ",get_xor_sbox(ckey, coffset))
        decrypt_raw = try_decrpt(ckey,coffset)
        if is_string(decrypt_raw[-0x40:]):
            print "  maybe : %x & %x" % (ckey, coffset)
            print decrypt_raw
```

Le résultat appliqué au fichier to_decrypt.txt :

```
decrypt.py to_decrypt.txt
  maybe : f6 & 0
Bs<┐r9^ö¯╬ôãCo<□ǂüA–♦ä_–ü▲&▒÷1çı¦ó
D«G£OÁ▄╬♥
h┼F╝┬z)c╩ƒ╦zy↔s×fqc_t0c═¥1═■ìÃ┼AðX¹êbí┴♫n-┼ã÷óÊh╬Ẅ4{º–ü^l§Ó↑ ÷x┼)l,├yd!>╝<-
]Uéoõ©lk¥¶y"Vjg"nÁiæTa♣m_¿¬kçoplyzêÔ÷t
´fs.Â^♦
ì╚Úääüì╚Ä Ü╚æìëÜ(~ÿâ×éàƒ×╩êÅâäì╩èıÅïõÜïèÔ╬ºÇ╬ÜåïÃTıò©éëô6¤êëôéfô■ÆjÎûÄòÆ~ÎâƒàÆûâ◄dçº±╟á¾í-▒╝¥¥Âç÷Ù÷▓¶
²Ó▓´³Ù▓¸³µ¹ty c=ncer<ed with its security to be proactive and to look for possible signs of
intrusion before having any proof.

Ho! You did it! Congratulation :-)

Don't forget to visit our website ;) https://exatrack.com/

Flag : ndh16_exatrackgiveyoumorepoints
```

C'est gagné ! Le flag était donc `ndh16_exatrackgiveyoumorepoints`. J'avais fait ce petit challenge pour allier du reverse engineering 64b avec un poil de désobfuscation et de la crypto (très) faible.

Merci d'avoir lu cette solution. Si vous l'avez cassé d'une autre façon (il y a plusieurs autres moyens) je serai ravi d'avoir votre retour :-)