

Challenge : CryptoLocker

Catégorie : Forensics

Énoncé :

Un de nos admins nous a appelé en urgence suite à un CryptoLocker qui s'est lancé sur un serveur ultra-sensible, juste après avoir appliqué une mise à jour fournie par notre prestataire informatique.

Ce *malware* vise spécifiquement un fichier pouvant faire perdre des millions d'euros à notre entreprise : il est très important de le retrouver !

L'administrateur nous a dit que pour éviter que le logiciel ne se propage, il a mis en pause le serveur virtualisé et a récupéré sa mémoire vive dès qu'il a détecté l'attaque.

Vous êtes notre seul espoir.

Fichier(s) : memory.dmp.gz

Table des matières

1)	KEY ET FLAG.ENC	2
2)	RANSOMWARE	3
3)	REVERSE ENGINEERING	4
A)	MAIN	4
B)	RECURSEFOLDER	4
C)	ENCRYPTFILE	6
4)	SCRIPT PYTHON	7

1) Key et flag.enc

Après décompression de l'archive, on prend rapidement quelques informations sur le dump mémoire fourni et on sélectionne le premier profil proposé par Volatility.

```
SoEasy in ~/Bureau [16:58]
> gunzip memory.dmp.gz

SoEasy in ~/Bureau [16:58]
> file memory.dmp
memory.dmp: MS Windows 32bit crash dump, PAE, full dump, 262030 pages

SoEasy in ~/Bureau [16:58]
> sudo volatility -f ./memory.dmp imageinfo
Volatility Foundation Volatility Framework 2.6
INFO : volatility.debug : Determining profile based on KDBG search...
Suggested Profile(s) : Win7SP1x86_23418, Win7SP0x86, Win7SP1x86_24000, Win7SP1x86 (Instantiated with WinXPS
P2x86)
AS Layer1 : IA32PagedMemoryPae (Kernel AS)
AS Layer2 : WindowsCrashDumpSpace32 (Unnamed AS)
AS Layer3 : FileAddressSpace (/root/Bureau/memory.dmp)
PAE type : PAE
DTB : 0x185000L
KUSER_SHARED_DATA : 0xffdf0000L
Image date and time : 2020-04-13 18:39:35 UTC+0000
Image local date and time : 2020-04-13 11:39:35 -0700
```

On utilisera donc ici le profil « Win7SP1x86_23418 ».

On commence alors l'énumération en faisant un scan des fichiers et on stocke le résultat dans un nouveau fichier dans lequel on approfondira nos recherches.

```
SoEasy in ~/Bureau [17:08]
> sudo volatility -f ./memory.dmp --profile=Win7SP1x86_23418 filescan > FILESCAN
Volatility Foundation Volatility Framework 2.6
```

On peut alors par exemple se rendre dans ce fichier nouvellement créé et rechercher le mot-clé (en tout cas en CTF) « flag ».

```
0x000000003ed13898 2 1 R-rw- \Device\HarddiskVolume1\Users\IEUser\Desktop\key.txt
0x000000003ed139f0 2 0 RW-rw- \Device\HarddiskVolume1\Users\IEUser\Desktop\flag.txt.enc
0x000000003ed13c88 10 1 RW-r-- \Device\HarddiskVolume1\Windows\System32\winevt\Logs\Microsoft-Windows-Appli
0x000000003ed145f0 1 1 R-r-d \Device\HarddiskVolume1\Windows\System32\en-US\FirewallAPI.dll.mui
0x000000003ed15220 2 0 RW-rwd \Device\HarddiskVolume1\Directory
0x000000003ed15710 1 1 R-r-- \Device\HarddiskVolume1\Windows\Registration\R0000000000006.clb
0x000000003ed15bf8 3 0 R-r-d \Device\HarddiskVolume1\Windows\System32\mfcsbss.dll
0x000000003ed16038 6 0 R-r-d \Device\HarddiskVolume1\Windows\System32\catsrv.dll
0x000000003ed178b0 8 0 R-r-d \Device\HarddiskVolume1\Windows\System32\en-US\wininit.exe.mui
0x000000003ed18c68 6 0 R-r-d \Device\HarddiskVolume1\Windows\System32\RpcRtRemote.dll
0x000000003ed193f8 7 0 R-r-d \Device\HarddiskVolume1\Windows\System32\VBBoxMRXNP.dll
0x000000003ed19f80 8 0 R-r-d \Device\HarddiskVolume1\Windows\System32\en-US\winlogon.exe.mui
0x000000003ed1aec8 4 0 R-r-d \Device\HarddiskVolume1\Windows\System32\drprov.dll
0x000000003ed1af80 8 0 R-r-- \Device\HarddiskVolume1\Windows\Fonts\batang.ttc
0x000000003ed1d2d8 8 0 R-r-- \Device\HarddiskVolume1\Windows\Fonts\msgothic.ttc
0x000000003ed1d988 8 0 R-r-- \Device\HarddiskVolume1\Windows\Fonts\gulim.ttc
0x000000003ed1e938 8 0 R-r-- \Device\HarddiskVolume1\Windows\System32\catroot\{F750E6C3-38EE-11D1-85E5-00
0x000000003ed1f308 8 0 R-r-- \Device\HarddiskVolume1\Windows\Fonts\meiryo.ttc
0x000000003ed1f4a0 8 0 R-r-- \Device\HarddiskVolume1\Windows\System32\catroot\{F750E6C3-38EE-11D1-85E5-00
0x000000003ed1f9b0 8 0 R-r-- \Device\HarddiskVolume1\Windows\Fonts\msjhbd.ttf
C'est la seule occurrence
```

^G Aide ^O Écrire ^W Chercher ^K Couper ^J Justifier ^C Pos. cur. M-U Annuler
 ^X Quitter ^R Lire fich. ^M Remplacer ^U Coller ^T Orthograp. ^_ Aller ligne M-E Refaire

On trouve alors une unique occurrence : un fichier « flag.txt.enc ».

De plus, on remarque aisément un fichier « key.txt » sur la ligne du dessus : nous allons alors chercher à dump ces deux fichiers en se servant de leurs noms et offsets.

```
SoEasY in ~/Bureau [17:25]
> volatility -f ./memory.dmp --profile=Win7SP1x86_23418 dumpfiles -Q 0x000000003ed139f0 --name flag.txt.enc -D .
Volatility Foundation Volatility Framework 2.6
DataSectionObject 0x3ed139f0 None \Device\HarddiskVolume1\Users\IEUser\Desktop\flag.txt.enc

SoEasY in ~/Bureau [17:25]
> volatility -f ./memory.dmp --profile=Win7SP1x86_23418 dumpfiles -Q 0x000000003ed13898 --name key.txt -D .
Volatility Foundation Volatility Framework 2.6
DataSectionObject 0x3ed13898 None \Device\HarddiskVolume1\Users\IEUser\Desktop\key.txt
SharedCacheMap 0x3ed13898 None \Device\HarddiskVolume1\Users\IEUser\Desktop\key.txt
```

```
SoEasY in ~/Bureau [17:26]
> mv file.None.0x854fbc98.key.txt.dat key.txt

SoEasY in ~/Bureau [17:27]
> mv file.None.0x855651e0.flag.txt.enc.dat flag.txt.enc
```

On peut ensuite faire un cat sur chacun des deux fichiers pour voir si le contenu est potentiellement exploitable.

```
SoEasY in ~/Bureau [17:30]
> cat flag.txt.enc
' {kp[U]
SUU ]Y^\TQU^UWR[W\QTPQ
^UQUVYZWQRWZP

SoEasY in ~/Bureau [17:35]
> cat key.txt
0ba883a22afb84506c8d8fd9e42a5ce4e8eb1cc87c315a28dd
```

On pourrait alors penser à quelque chose comme du SHA1 ou SHA256 mais la longueur de la clé ne correspond pas.

2) Ransomware

L'énoncé va ensuite nous être utile : on va chercher dans notre dump des fichiers un programme qui serait susceptible d'être assimilé à un « CryptoLocker ».

Comme nous sommes sur Windows, on peut alors affiner la recherche pour afficher les exécutables (fichiers « .exe »). Après un moment de recherche, un fichier retient enfin notre attention.

```
0x000000003ed66b60 6 0 R--r-d \Device\HarddiskVolume1\Users\IEUser\Desktop\update_v0.5.exe
```

En effet, on retrouve ici un exécutable qui se trouve sur le bureau de l'utilisateur. De plus, son nom, « update_v0.5 » rappelle une technique classique utilisée par les pirates : faire croire à une mise à jour d'un logiciel pour faire exécuter le malware par une cible (d'autant plus que le fichier se nomme « update » sans spécifier le logiciel mis à jour...).

On peut alors extraire cet exécutable (puis le renommer) pour l'analyser.

```
SoEasY in ~/Bureau [18:10]
> volatility -f ./memory.dmp --profile=Win7SP1x86_23418 dumpfiles -Q 0x000000003ed66b60 --name -D .
Volatility Foundation Volatility Framework 2.6
ImageSectionObject 0x3ed66b60 None \Device\HarddiskVolume1\Users\IEUser\Desktop\update_v0.5.exe
DataSectionObject 0x3ed66b60 None \Device\HarddiskVolume1\Users\IEUser\Desktop\update_v0.5.exe

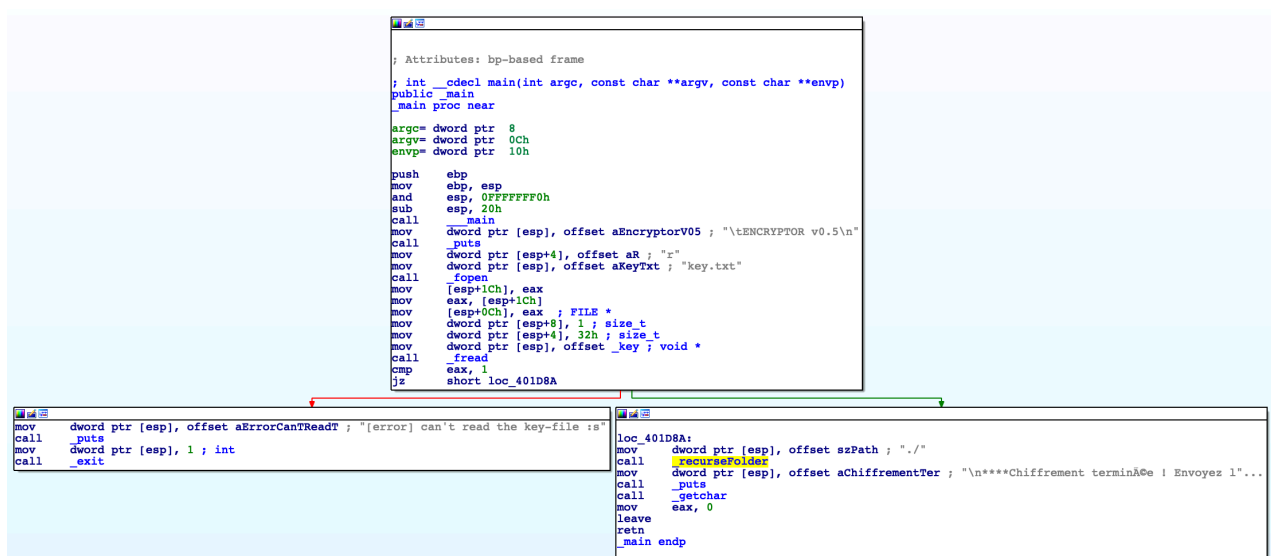
SoEasY in ~/Bureau [18:10]
> mv file.None.0x85279860.update_v0.5.exe.dat update_v0.5.exe
```

3) Reverse Engineering

a) Main

Une fois le binaire extrait, on peut commencer le Reverse Engineering avec une analyse statique du code : j'utiliserais ici le désassembleur IDA (ainsi que le decompiler HexRays associé).

Comme d'habitude on commence par désassembler la fonction main, et on observe en premier l'affichage de la chaîne « ENCRYPTOR v0.5 », qui explique donc le nom de l'exécutable.



```

; Attributes: bp-based frame
; int __cdecl main(int argc, const char **argv, const char **envp)
public _main
_main proc near
    argc= dword ptr 8
    argv= dword ptr 0Ch
    envp= dword ptr 10h

    push    ebp
    mov     ebp, esp
    and     esp, 0FFFFFFFh
    sub     esp, 20h
    call    _main
    mov     dword ptr [esp], offset aEncryptorV05 ; "\tENCRYPTOR v0.5\n"
    call    puts
    mov     dword ptr [esp+4], offset aR ; "r"
    mov     dword ptr [esp], offset aKeyTxt ; "key.txt"
    call    fopen
    mov     [esp+1Ch], eax
    mov     eax, [esp+1Ch]
    mov     [esp+0Ch], eax ; FILE *
    mov     dword ptr [esp+8], 1 ; size_t
    mov     dword ptr [esp+4], 32h ; size_t
    mov     dword ptr [esp], offset _key ; void *
    call    fread
    cmp     eax, 1
    jz      short loc_401D8A

loc_401D8A:
    mov     dword ptr [esp], offset szPath ; "/"
    call    _recurseFolder
    mov     dword ptr [esp], offset aChiffrementTer ; "\n****Chiffrement termin   ! Envoyez 1"...
    call    puts
    call    _getchar
    mov     eax, 0
    leave
    retn
_main endp
  
```

On voit ici que le programme va ouvrir le fichier « key.txt » en mode lecture avec l'appel à la fonction fopen.

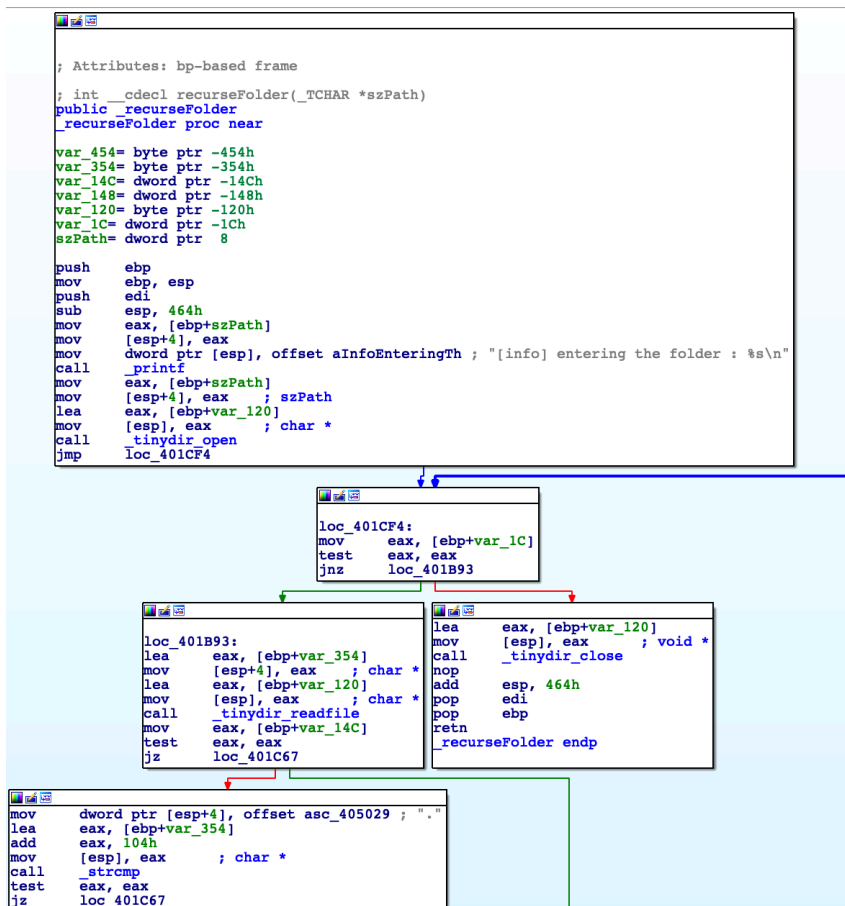
Si le fichier n'est pas trouvé, (jump de gauche sur le graphe) le message d'erreur « [error] can't read the key-file :s ».

Si le fichier est bien trouvé, la fonction « _recurseFolder » est appelée, suivie du message « *****Chiffrement terminé ! Envoyez l'argent ! ».

b) RecurseFolder

La fonction qui va nous intéresser ici sera donc la fonction appelée avant le message signifiant le fin du chiffrement des fichiers par le CryptoLocker : il s'agit de la fonction « _recurseFolder ».

Cette fonction a pour but de parcourir l'arborescence de la machine de la victime, récursivement, à la recherche du dossier contenant le fichier « flag.txt » (d'où le nom de la fonction).



On va alors chercher à savoir ce que le programme va effectuer comme instructions lorsque le fichier « flag.txt » sera trouvé.



On voit ici que si le programme trouve un fichier du nom de « flag.txt » (sort du strcmp avec EAX à zéro et ne prend pas le jump vers « loc_401CE6 ») alors le message « [info] file encryptable found : flag.txt » est affiché suite à quoi la fonction « _encryptFile » est appelée.

c) EncryptFile

La fonction qui va finalement nous intéresser se trouve donc être « _encryptFile ».

Des informations importantes apparaissent au début de la fonction : le programme va ouvrir le fichier « flag.txt » en mode lecture binaire (fopen('flag.txt', 'rb')), et ouvrir un fichier en mode écriture binaire avec la possibilité de créer le fichier s'il n'existe pas auparavant : ce sera notre fichier « flag.txt.enc » (fopen('flag.txt.enc', 'wb+')).

```

push    ebp
mov     ebp, esp
push    edi
push    ebx
sub     esp, 130h
mov     eax, [ebp+arg_0]
mov     [esp+4], eax ; char *
lea     eax, [ebp+var_128]
mov     [esp], eax ; char *
call    _strcpy
lea     eax, [ebp+var_128]
mov     ecx, 0FFFFFFFh
mov     edx, eax
mov     eax, 0
mov     edi, edx
repne scasb
mov     eax, ecx
not     eax
lea     edx, [eax-1]
lea     eax, [ebp+var_128]
add     eax, edx
mov     dword ptr [eax], 636E652Eh
mov     byte ptr [eax+4], 0
mov     dword ptr [esp+4], offset aRb ; "rb"
mov     eax, [ebp+arg_0]
mov     [esp], eax ; char *
call    fopen
mov     [ebp+var_14], eax
mov     dword ptr [esp+4], offset aWb ; "wb+"
lea     eax, [ebp+var_128]
mov     [esp], eax ; char *
call    _fopen

```

On remarque ensuite un bloc d'instructions intéressant qui va faire appel au fichier « key.txt » ouvert au début de l'exécution du programme et effectuer entre autres un XOR : on se trouve bien dans la partie intéressante du code.

```

loc_401AB7:
mov     edx, [ebp+var_C]
mov     eax, [ebp+var_20]
add     eax, edx
movzx   ebx, byte ptr [eax]
mov     eax, [ebp+var_C]
add     eax, 2
mov     edx, 0
div     [ebp+var_24]
mov     eax, edx
movzx   ecx, ds:key[eax]
mov     edx, [ebp+var_C]
mov     eax, [ebp+var_20]
add     eax, edx
xor     ebx, ecx
mov     edx, ebx
mov     [eax], dl
add     [ebp+var_C], 1

```

On peut cette fois utiliser le décompilateur HexRays livré avec IDA Pro pour avoir un aperçu rapide du code de cette fonction et comprendre le fonctionnement global plus efficacement.

```
int __cdecl encryptFile(char *a1)
{
    char *v1; // eax
    char v3[256]; // [esp+10h] [ebp-128h]
    size_t v4; // [esp+110h] [ebp-28h]
    size_t v5; // [esp+114h] [ebp-24h]
    void *v6; // [esp+118h] [ebp-20h]
    size_t v7; // [esp+11Ch] [ebp-1Ch]
    FILE *v8; // [esp+120h] [ebp-18h]
    FILE *v9; // [esp+124h] [ebp-14h]
    int j; // [esp+128h] [ebp-10h]
    int i; // [esp+12Ch] [ebp-Ch]

    strcpy(v3, a1);
    v1 = &v3[strlen(v3)];

    *(_DWORD *)v1 = 1668179246;
    v1[4] = 0;

    v9 = fopen(a1, "rb");
    v8 = fopen(v3, "wb+");

    fseek(v9, 0, 2);
    v7 = ftell(v9);
    v6 = malloc(v7);
    fseek(v9, 0, 0);

    v5 = strlen(key);
    v4 = fread(v6, 1, v7, v9);

    for ( i = 0; i < (signed int)v7; ++i )
        *(_BYTE *)v6 + i ^= key[(i + 2) % v5];

    for ( j = 0; j < (signed int)v7; ++j )
        putc(*((char *)v6 + j), v8);

    free(v6);
    fclose(v9);
    fclose(v8);
    return remove(a1);
}
```

Pour synthétiser cette fonction, on a donc :

```
for(int i=0; i<flag.length; i++)
    flag_enc[i] = flag[i] ^ key[i + 2 % 50]
```

4) Script Python

On peut ainsi écrire un script en python pour inverser le processus.

Je fais ici le choix de passer les fichier « key.txt » ainsi que « flag.txt.enc » en arguments au programme (une envie soudaine) : je commence donc par vérifier que le nombre d'arguments entrés à l'appel du programme est égal (ou supérieur) à 2 et j'importe la librairie « sys ».

Si c'est le nombre d'arguments est correct, je vais appeler ma fonction principale nommée « get_flag » (en lui laissant la possibilité d'exploiter elle aussi les arguments passé en paramètres au programme). Sinon, j'affiche un message qui indique l'utilisation du programme.

```
1 import sys
```

```
24 if(len(sys.argv) < 2):
25     print("[!] Usage : python CryptoLock.py <key file> <encrypted flag file>")
26 else:
27     get_flag(sys.argv)
```

Le premier argument correspondra donc à la clé et le deuxième argument au flag chiffré. Il nous faut tout d'abord connaître la taille de ces deux éléments : on va pour cela xxd chacun des deux fichiers et relever leur taille en octets.


```
SoEasY in ~ [21:03]
> xxd Bureau/flag.txt.enc
00000000: 277b 6b70 1a01 0055 0507 5d0c 5355 0555  '{kp ... U.. ].SU.U
00000010: 095d 595e 065c 0402 0654 0751 0055 015e  .]Y^.\ ... T.Q.U.^
00000020: 5557 525b 575c 5154 5007 5107 0b5e 5551  UWR[W\QTP.Q..^UQ
00000030: 5556 0259 5a07 0502 5751 5201 0f03 5702  UV.YZ ... WQR ... W.
00000040: 0601 5a50 0f1b 6e00 0000 0000 0000 0000  .. ZP .. n.....
00000050: 0000 0000 0000 0000 0000 0000 0000 0000  .....
00000060: 0000 0000 0000 0000 0000 0000 0000 0000  .....
```

```
SoEasY in ~ [21:03]
> xxd Bureau/key.txt
00000000: 3062 6138 3833 6132 3261 6662 3834 3530  0ba883a22afb8450
00000010: 3663 3864 3866 6439 6534 3261 3563 6534  6c8d8fd9e42a5ce4
00000020: 6538 6562 3163 6338 3763 3331 3561 3238  e8eb1cc87c315a28
00000030: 6464 0000 0000 0000 0000 0000 0000 0000  dd.....
00000040: 0000 0000 0000 0000 0000 0000 0000 0000  .....
00000050: 0000 0000 0000 0000 0000 0000 0000 0000  .....
00000060: 0000 0000 0000 0000 0000 0000 0000 0000  .....
```

Le reste des fichiers étant constitué uniquement de zéros qui n'auront aucune influence sur le calcul, on peut réduire la taille de la clé à 50 octets et celle du flag chiffré à 70 octets.

La subtilité ici repose dans le fait d'ouvrir en mode binaire les fichiers, c'est à ne pas oublier. On définit ainsi nos deux variables « key » et « flag_enc ».

```
3 def get_flag(argv):
4     key=[]
5     print("\n[+] Ouverture du fichier", sys.argv[1])
6     with open(sys.argv[1], 'rb') as k:
7         for i in range(50):
8             key.append(k.read(1))
9
10    flag_enc=[]
11    print("[+] Ouverture du fichier", sys.argv[2])
12    with open(sys.argv[2], 'rb') as f:
13        for i in range(70):
14            flag_enc.append(f.read(1))
15
```

On définit ensuite une variable « flag » dans laquelle on stockera le résultat de l'inverse de l'opération qui synthétisait la fonction « _encryptFile » énoncée plus tôt et on finira par afficher ce flag.

```
16    flag=[]
17    print("[+] Dechiffrement du flag")
18    for i in range(len(flag_enc)):
19        char = int.from_bytes(flag_enc[i], byteorder='big') ^ int.from_bytes(key[(i+2)%len(key)], byteorder='big')
20        flag.append(chr(char))
21
22    print("[+] Flag :", "".join(flag))
23
```

Tout ceci nous donne le script au complet.


```

1  import sys
2
3  def get_flag(argv):
4      key=[]
5      print("\n[+] Ouverture du fichier", sys.argv[1])
6      with open(sys.argv[1], 'rb') as k:
7          for i in range(50):
8              key.append(k.read(1))
9
10     flag_enc=[]
11     print("[+] Ouverture du fichier", sys.argv[2])
12     with open(sys.argv[2], 'rb') as f:
13         for i in range(70):
14             flag_enc.append(f.read(1))
15
16     flag=[]
17     print("[+] Dechiffrement du flag")
18     for i in range(len(flag_enc)):
19         char = int.from_bytes(flag_enc[i], byteorder='big') ^ int.from_bytes(key[(i+2)%len(key)], byteorder='big')
20         flag.append(chr(char))
21
22     print("[+] Flag :", "".join(flag))
23
24 if(len(sys.argv) < 2):
25     print("[!] Usage : python CryptoLock.py <key file> <encrypted flag file>")
26 else:
27     get_flag(sys.argv)

```

On peut alors exécuter le script avec les bons paramètres.

```

SoEasy in ~/Bureau [21:24]"
> python3 CryptoLock.py key.txt flag.txt.enc

[+] Ouverture du fichier key.txt
[+] Ouverture du fichier flag.txt.enc
[+] Dechiffrement du flag
[+] Flag : FCSC{324cee8fe3619a8bea64522eadf05c84df7c6df9f15e4cab4d0e04c77b20bb47}

```

On récupère ainsi le flag ! Encore un challenge très sympathique.